

普通高等教育“十一五”国家级规划教材
微电子与集成电路设计系列规划教材

嵌入式系统

——从 SoC 芯片到系统

（第 2 版）

凌 明 王学香 单伟伟 编著

电子工业出版社
Publishing House of Electronics Industry
北京·BEIJING

内 容 简 介

本书是普通高等教育“十一五”国家级规划教材，从 SoC 芯片设计者的视角分析和介绍嵌入式系统的基础知识，注重软硬件协同设计与软硬件适配优化。全书共 8 章，主要内容包括：嵌入式系统概况、嵌入式系统中 SoC 的硬件架构、嵌入式系统的开发和调试、SoC 中的 CPU 内核、存储子系统、外设接口、嵌入式系统软件概述、嵌入式系统功耗优化等。每章都设置专门的设计案例分析、思考题和扩展阅读，配套电子课件、习题参考答案、程序代码等。

本书可作为高等学校微电子、集成电路设计，以及其他电子和计算机相关专业高年级本科生与研究生 SoC 及嵌入式课程的教材，也可作为从事嵌入式系统研究开发的工程技术人员的参考书。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目 (CIP) 数据

嵌入式系统：从 SoC 芯片到系统 / 凌明，王学香，单伟伟编著. — 2 版. — 北京：电子工业出版社，2017.6
ISBN 978-7-121-30718-8

I. ①嵌… II. ①凌… ②王… ③单… III. ①微型计算机—系统开发—高等学校—教材 IV. ①TP360.21

中国版本图书馆 CIP 数据核字 (2016) 第 314602 号

策划编辑：王羽佳

责任编辑：周宏敏

印 刷：

装 订：

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编：100036

开 本：787×1092 1/16 印张：26.5 字数：773 千字

版 次：2017 年 6 月第 1 版

印 次：2017 年 6 月第 1 次印刷

印 数：3000 册 定价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888，88258888。

质量投诉请发邮件至 zltz@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：(010) 88254535，wyj@phei.com.cn。

第 2 版 序

《嵌入式系统——基于 SEP3203 微处理器的应用开发》第 1 版自 2006 年底出版以来得到了东南大学以及相关兄弟院校的教学应用，取得了良好的教学效果，并入选了“十一五”国家级规划教材。近几年来，嵌入式系统技术的飞速发展正推动着整个产业发生着深刻的变化：第一，移动互联网产业的兴起彻底改变了传统电信运行商、软件服务商、内容提供商、整机制造商等生态环境的商业模式，而物联网技术的迅速发展则推动着传统产业信息化和现代服务业的变革；第二，嵌入式领域虽然不存在传统桌面系统的 Wintel 联盟，但是在嵌入式 CPU 和嵌入式操作系统方面已经出现 ARM 公司系列 CPU 和 Google 公司 Android 系统一枝独秀的局面；第三，国内已经涌现出一批初具规模和实力的嵌入式微处理器设计厂商，自主 CPU 内核也逐渐成熟，并在部分市场领域得到应用，自主 SoC 芯片的生态环境已初步形成。

另一方面，嵌入式系统的本质是以应用为核心的专用计算机系统，它的核心是嵌入式微处理器（SoC 芯片），围绕应用目标开展软硬件协同设计是嵌入式技术的本质特征。作为专用计算机系统，嵌入式系统技术与通用计算机系统没有本质的区别，但往往对系统的性能、功耗、成本、可靠性与实时性有更加严格的限制，这就决定了其总线架构、CPU 内核设计、存储架构设计、媒体与图形系统设计等方面都有自己的设计哲学。传统的嵌入式系统教学往往采用国外 SoC 芯片作为主要教学内容，由于缺乏芯片的底层实现细节，往往更侧重于如何基于国外芯片厂商封装好的底层软件库进行应用开发。这使得学生缺乏对芯片底层工作机制的了解，从而无法对软硬适配部分进行充分的优化与定制，更不要说参与自主 SoC 芯片的研发与系统方案设计了。这显然无法满足技术和产业发展的需求。

基于对以上问题的认识，东南大学国家 ASIC 工程技术研究中心（以下简称工程中心）的几位老师着手编写了《嵌入式系统——从 SoC 芯片到系统（第 2 版）》。正如本书副标题所表明的，本书将从 SoC 芯片的视角来分析和介绍嵌入式系统的基础知识。得益于工程中心十多年来在自主 SoC 芯片设计方面积累的工程经验与研究成果，本书详细介绍嵌入式微处理器 SoC 芯片的架构、CPU、存储子系统、外围设备、调试方法、低功耗设计和操作系统基础，力图给读者一个 SoC 芯片设计者的视角，使其了解现代嵌入式系统的构成，并真正理解一个复杂的嵌入式系统应用在芯片层面和软件层面是如何运作的。在第 2 版中，作者将继续采用工程中心自主研发的 SEP4020 与 SEP6200 处理器作为主要案例，其中 SEP6200 处理器采用了国产 UniCore2 CPU 内核。

与第 1 版不同，为了便于教学，本书为几乎所有的章节编写了思考题、案例分析和扩展阅读。当然，这不是一本包罗万象、解决所有问题的教材，正如作者们所希望的，这是一本嵌入式系统入门教材。通过这本书的阅读和学习，作者希望为读者后续进一步的深入学习打下坚实的基础。

时龙兴
于东南大学

前 言

关于本书以及如何使用本书

毫无疑问，在大学中讲授嵌入式系统相关的课程是一个巨大的挑战。带来这种挑战的根本原因在于：第一，嵌入式系统是一个飞速发展的动态系统，新技术、新产品、新应用、新商业模式层出不穷，这一点在这一轮的移动互联网浪潮和物联网浪潮中表现得尤为突出。第二，嵌入式系统技术是一个非常综合的学科门类，其内涵涉及电子与微电子、计算机架构、操作系统、中间件、人机交互、计算机网络、通信甚至整机设计与制造等各领域，很难在有限的课时内将所有的知识点讲深讲透——这一点对于传统的电子信息类专业的本科生而言尤其明显，因为传统的电子信息类专业课程的设置往往很少涉及操作系统等软件方面的内容。第三，嵌入式系统技术的两大基础是微电子技术（尤其是 SoC 技术）和以嵌入式操作系统为代表的嵌入式软件技术。嵌入式系统对于性能、成本、功耗、实时性与高可靠性的更高要求决定了围绕应用目标开展软硬件协同设计和软硬件适配优化是嵌入式系统技术的本质特征。这就需要对 SoC 芯片的架构和工作机制有非常深刻的理解。第四，嵌入式系统技术是一门实践性非常强的课程，如何在有限的课时内加强学生的实践能力培养也是一个非常大的挑战。

针对上述这些挑战，东南大学国家专用集成电路系统工程技术研究中心依托电子科学与技术国家重点学科的优势，将多年来承担的国家、省部级科研项目的科研成果应用于教学，开展软硬件协同的嵌入式系统教学，具体举措包括：第一，建设了以自主 SoC 芯片为基础的系列嵌入式系统课程：面向本科生的“嵌入式系统概论”，面向研究生课程的“嵌入式系统”、“SoC 设计”、“嵌入式系统高级 C 语言编程”、“嵌入式操作系统”、“移动互联网应用编程”、“嵌入式系统实训”。课程内容覆盖从 SoC 芯片到软硬协同设计、适配优化再到应用方案设计的各层次。第二，采用自主 SoC 芯片设计了多款教学实验平台，并开发了相关实验。在课内有限的学时内重点培养学生基本的实践技能，通过课外实验和综合实训课程培养学生的综合应用能力，通过连续举办多届嵌入式设计竞赛培养学生的创新能力。

本书的写作目的就是配合本科课程“嵌入式系统概论”和研究生课程“嵌入式系统”的教学。因为是作为本科概论课程和研究生基础课程用书，本书的目的不是也不可能是讲授嵌入式系统相关的内容。事实上，试图在一本书、一门课程中将嵌入式系统的所有内容讲授清楚是不可能的。因此，**本书的基本定位是基础与入门**。为后续课程或进一步的学习打下坚实的基础和扫清概念障碍。我们发现初学者对于现代 SoC，尤其是面向移动互联网终端的 SoC 中出现的新概念、新知识的了解非常匮乏，而这些对于开展软硬件协同设计与软硬件适配优化却是必不可少的。与以往许多教材不同，本书试图站在 SoC 设计者的角度来介绍嵌入式系统的相关基础知识。这得益于本书作者参与了东南大学国家 ASIC 工程中心几款自主 SoC 芯片设计的全过程，全书中我们也将以**自主 SoC 芯片 SEP4020 和 SEP6200 作为案例进行介绍**。

为了便于教学，本书几乎在所有的章节中都设置了专门的设计案例和思考题。为了适应研究生教学的需要，本书在每章都设置了提高内容（书中带*的章节），将最新的技术发展融入教学内容。这些内容涉及到新型的片上互联架构、高性能 CPU 架构、异构计算单元、片上存储架构、多层次低功耗设计等内容，并为研究生和学有余力的本科生设置了扩展阅读单元，读者们可以通过这些扩展阅读进一步深化对相关技术的了解。

本书各章节安排如下：

第 1 章 嵌入式系统概况

本章概述嵌入式系统的概念、嵌入式系统的应用等。作为两个最重要也可能是最热门的应用领域，本章专门对移动互联网终端和物联网进行介绍。我们还将在本章介绍嵌入式系统的产业链构成，并对学习嵌入式系统的知识体系进行梳理。本章最后通过一个 MP3 播放器的实例分析引出问题：按下按钮后 MP3 音乐是怎么播放出来的。

第 2 章 嵌入式系统中 SoC 的硬件架构

本章介绍 SoC 的硬件架构，由总线互联、处理器、中断控制器、定时器、存储系统、外设接口等组成，由于本书后面的章节将详细阐述处理器、存储系统、各种接口等部分，因此本章重点介绍 SoC 芯片的总线互联，并分析 ARM 公司的片上互联标准——AMBA。本章还专门介绍 DMA 控制器的工作原理。在案例部分，本章给出了几款 SoC 芯片的架构介绍。

第 3 章 嵌入式系统的开发和调试

本章首先介绍嵌入式系统的一般开发过程，并对各种调试方法进行介绍，包括模拟器、在线仿真、片上在线仿真和跟踪技术，并详细介绍基于 JTAG 的调试方法。接着对 ARM 公司的 3 款集成开发环境：ADS、MD-5 和 MDK 进行介绍。

第 4 章 SoC 中的 CPU 内核

本章介绍 CPU 的基本概念：CPU 的流水线技术、分支预测技术、乱序执行技术、超标量处理器、VLIW 处理器、EPIC 处理器、多核处理器等。接下来重点介绍 ARM 处理器的特点、编程模型、指令系统、汇编语言程序结构、异常处理、编程技巧、ARM 系列微处理器等。作为对 ARM 架构的补充，本章还介绍 MIPS、龙芯和众志 CPU 的基本架构。GPU 和可重构计算作为新兴的计算引擎也在本章进行介绍。

第 5 章 存储子系统

本章首先介绍嵌入式系统存储子系统的金字塔结构，重点分析高速缓存（Cache）与虚拟存储器技术。接下来介绍常用的存储器及其时序，包括 SRAM、SDRAM、DDR SDRAM、Flash 等。SoC 中的外部存储器控制接口 EMI 及其编程模型，SoC 中的 SD/MMC 控制器及其编程模型也在本章进行讲解。最后介绍存储子系统的性能和功耗优化技术。

第 6 章 外设接口

本章分别介绍低速通信接口控制器（异步串行通信 UART、同步串行通信 SPI）、高速通信接口控制器（通用串行总线 USB、10/100M 以太网 MAC 网络接口）、人机界面控制器（液晶显示器控制器 LCDC、音频接口 I2S 控制器）。

第 7 章 嵌入式系统软件概述

本章首先介绍嵌入式系统软件的组成、bootloader 引导程序，以及为什么需要嵌入式操作系统和嵌入式操作系统的一些基本概念。在本章的第二部分，重点介绍与嵌入式操作系统相关的基本原理，包括内核中的任务管理、任务间通信和中断管理。Android 作为当前最流行的手持终端操作系统在本章的第三部分进行介绍。

第8章 嵌入式系统功耗优化

本章首先概述低功耗设计方法。接着分别介绍 SoC 级低功耗设计方法和嵌入式系统级低功耗设计方法，并给出两个与之对应的设计案例。

相应的课程安排

本书可以作为电子类本科生高年级和研究生低年级的教材。按照 2 学分（本科 32 学时、研究生 36 学时，其中实验课时按一半折算）的课程设置，作者给出的参考课时安排如下（教师可根据实际情况进行调整）：

- 本科生“嵌入式系统概论”授课学时安排（24 学时）

- 第 1 章 嵌入式系统概况 2 学时

- 第 2 章 嵌入式系统中 SoC 的硬件架构 2 学时

- 第 3 章 嵌入式系统的开发和调试 2 学时

- 第 4 章 SoC 中的 CPU 内核 6 学时

- 第 5 章 存储子系统 3 学时

- 第 6 章 外设接口 3 学时

- 第 7 章 嵌入式系统软件概述 3 学时

- 第 8 章 嵌入式系统功耗优化 3 学时

- 本科生“嵌入式系统概论”实验课学时安排（16 学时，折合 8 学时）

- 实验一 UB4020EVB 实验平台及 ADS 开发环境及 ARM 汇编实验 3 学时

- 实验二 GPIO、Timer（含中断和逻辑分析仪使用） 3 学时

- 实验三 UART、IIC 与 SPI 实验（逻辑分析仪） 3 学时

- 实验四 DMA 与 LCDC 实验 3 学时

- 实验五 设计一个温湿度传感节点 4 学时

- 本科生“嵌入式系统概论”课外实验安排

- 1. Mini4020 或 HiveBoard 教学平台相关 Linux 系统实验

- 研究生“嵌入式系统”授课学时安排（28 学时）

- 第 1 章 嵌入式系统概况 1 学时

- 第 2 章 嵌入式系统中 SoC 的硬件架构 6 学时

- 第 3 章 嵌入式系统的开发和调试（自学）

- 第 4 章 SoC 中的 CPU 内核 6 学时

- 第 5 章 存储子系统 6 学时

- 第 6 章 外设接口（自学）

- 第 7 章 嵌入式系统软件概述 3 学时

- 第 8 章 嵌入式系统功耗优化 6 学时

- 研究生“嵌入式系统”实验课学时安排（16 学时，折合 8 学时）

- 实验一 UB4020EVB 实验平台及 ADS 开发环境及 ARM 汇编实验 3 学时

- 实验二 GPIO、Timer、UART 实验（含中断和逻辑分析仪使用） 3 学时

- 实验三 不同 CPU 主频、总线主频、DDR 控制器主频对于性能的影响 3 学时

- 实验四 采用 Oprofile 分析软件瓶颈 3 学时

- 实验五 SoC 中多总线 Master 竞争对于系统性能的影响 4 学时

*本实验课程需要采用 UB4020EVB 开发板和 HiveBoard 开发板。

● 研究生“嵌入式系统”课外实验安排

1. Mini4020 教学平台相关 Linux 系统实验

2. HiveBoard 教学平台相关 Linux 系统实验及其他高阶实验

支持资料

本书中的许多图表可以无偿地从互联网上得到但不得用于商业目的。使用它们的唯一限制是使用这些资料的任何课程都应将本书作为推荐教材。

作者已做了很大的努力来校验本书内容，相关的其他人员更是以加倍的努力来校验这本书。即便如此，也仍然可能会遗漏某些错误。另外，由于作者水平与知识面的限制，书中的部分观点与见解可能存在欠妥甚至错误之处，作者欢迎读者对本书的内容和形式以及发现的任何错误给予反馈意见。请将相关信息发至 E-mail: wxx@seu.edu.cn 和 trio@seu.edu.cn。

致谢

东南大学国家专用集成电路系统工程技术研究中心主任时龙兴教授参与了本书的策划和章节设置讨论，并给与了大量而具体的指导意见。东南大学国家专用集成电路系统工程技术研究中心的博士生张阳同学和谢震同学在本书的编写过程中也参与了讨论并给予了相应的技术支持。东南大学苏州研究院国家专用集成电路与系统实验室的硕士研究生刘烁、高滔、骆然、郑晓萌、刘克桥、李晨锋、蔡鹏翔、戴悦等参与了本书的部分图表绘制和勘误工作。在本书的写作与修改成稿期间，我正在加拿大维多利亚大学做访问学者，维大计算机系的潘建平教授和郑开明（Mantis Cheng）教授在工作和生活方面都给了我巨大的帮助。另外，本书得以最终出版，也得益于电子工业出版社的王羽佳编辑的大力支持与耐心等待。在此一并向他们表示衷心的感谢！

凌 明

目 录

第 1 章 嵌入式系统概况	1	思考题	47
1.1 什么是嵌入式系统	1	扩展阅读	47
1.2 嵌入式系统的应用与分类	2	第 3 章 嵌入式系统的开发和调试	48
1.2.1 基于实时性的分类	2	3.1 嵌入式系统的一般开发过程	48
1.2.2 基于应用的分类	2	3.1.1 交叉编译	49
1.2.3 移动互联网	3	3.1.2 链接	50
1.2.4 物联网	4	3.1.3 调试	50
1.3 嵌入式系统的产业链	6	3.2 调试方式介绍	51
1.4 嵌入式系统的知识体系	7	3.2.1 模拟器	53
1.5 案例: MP3 播放器	9	3.2.2 驻留监控软件	60
思考题	10	3.2.3 在线仿真调试	62
扩展阅读	11	3.2.4 片上在线仿真调试	62
第 2 章 嵌入式系统中 SoC 的硬件架构	12	3.2.5 跟踪 (Trace) 技术	63
2.1 SoC 硬件架构概述	12	3.2.6* CoreSight 调试与跟踪技术简介	65
2.2 互联结构	15	3.3 基于 JTAG 接口的片上在线仿真	70
2.2.1 常见互联结构分类	15	3.3.1 JTAG 简介	70
2.2.2 地址空间	20	3.3.2 基于 JTAG 的片上在线仿真的 系统结构	75
2.2.3 常见互联结构接口协议	23	3.3.3* ARM7TDMI 内核调试原理	76
2.3 中央处理器	32	3.4 ARM 的集成开发环境	82
2.4 中断控制器	32	3.4.1 ADS 集成开发环境	83
2.5 存储子系统	34	3.4.2 DS-5 集成开发环境	83
2.6 直接存储器访问 (DMA)	35	3.4.3 MDK 集成开发环境	89
2.6.1 scatter-gather DMA	36	3.5 嵌入式软件的执行镜像与启动过程	94
2.6.2 SEP4020 芯片中的 DMA 控制器	37	3.5.1 ARM 链接器的输出文件的加载 视图与执行视图	95
2.6.3 DMAC 驱动	40	3.5.2 基于 ROM 的程序执行	97
2.7 外设接口控制器	40	3.5.3 基于 RAM 的程序执行	97
2.7.1 高速通信接口控制器	40	3.5.4 ROM/RAM 重映射	98
2.7.2 低速通信接口控制器	41	思考题	98
2.7.3 人机界面控制器	41	扩展阅读	98
2.8 案例: SoC 架构设计	41	第 4 章 SoC 中的 CPU 内核	100
2.8.1 S3C44B0X	41	4.1 CPU 的基本概念	100
2.8.2 S3C6410	42	4.1.1 CPU 的发展	100
2.8.3 OMAP3530	43		
2.8.4 SEP4020	43		
2.8.5 SEP6200	44		

4.1.2 复杂指令集 (CISC) 与精简指令集 (RISC)	103	5.3.2 虚实地址映射与转换	224
4.1.3 CPU 的流水线技术	104	5.3.3 快速地址转换技术	227
4.1.4* CPU 的分支预测技术	106	5.3.4 地址保护机制	228
4.1.5* 乱序超标量处理器	110	5.3.5 处理缺页和 TLB 缺失	230
4.1.6* SIMD 和向量处理器	114	5.3.6 ARM Cortex A 系列处理器的 虚存管理	230
4.1.7* VLIW 处理器	115	5.4 片外存储器	234
4.1.8* EPIC 处理器	116	5.4.1 静态随机存储器 (SRAM)	235
4.2 ARM 内核	116	5.4.2 动态随机存储器 (DRAM)	237
4.2.1 ARM 介绍	116	5.4.3 非易失性存储器	250
4.2.2 ARM7TDMI 编程模型	121	5.5 外部存储器接口	258
4.2.3 ARM7TDMI 的指令集	131	5.5.1 SEP4020 芯片的外部存储器 接口 EMI	258
4.2.4 ARM7TDMI 汇编语言	143	5.5.2 SEP4020 芯片 EMI 的初始化与 配置	261
4.2.5 ARM7TDMI 异常处理	146	5.5.3 OMAP4460 处理器的外部 存储器接口	266
4.2.6 ARM 汇编程序与 C 程序	153	5.6* 存储子系统优化技术	267
4.2.7* ARM 处理器的多核技术	158	5.6.1 存储子系统的技术指标	267
4.2.8* ARM 处理器的最新发展	164	5.6.2 DDR 控制器的优化	271
4.3* 其他 CPU 介绍	169	5.6.3 片上存储器布局优化技术	276
4.3.1 MIPS 体系架构	170	案例: 高效高清媒体处理器的访存 QoS	279
4.3.2 龙芯处理器	175	思考题	284
4.3.3 UniCore-2 处理器	178	扩展阅读	285
4.4* 其他类型的计算引擎	181	第 6 章 外设接口	288
4.4.1* GPU	181	6.1 低速通信接口	288
4.4.2* 可重构计算	187	6.1.1 异步串行通信 UART	288
案例: REMUS-II 粗粒度可重构计算 架构	196	6.1.2 同步串行通信	294
思考题	201	6.2 高速通信接口	299
扩展阅读	202	6.2.1 通用串行总线 USB	299
第 5 章 存储子系统	203	6.2.2* 10/100M 以太网 MAC 网络 接口	304
5.1 存储子系统概述	203	6.3 人机接口	313
5.2 高速缓存 Cache	204	6.3.1 液晶显示器接口	313
5.2.1 Cache 的基本组成	204	6.3.2 音频接口	322
5.2.2 Cache 的基本原理	206	6.3.3 触摸屏接口	326
5.2.3* Cache 缺失与访问冲突	212	6.4 定时器	332
5.2.4* Cache 一致性问题	216	6.4.1 通用定时器	332
5.2.5 Cache 和 SPM 的比较	218	6.4.2 RTC	333
5.2.6* ARM Cortex A8 处理器的 Cache 架构	221		
5.3 虚拟存储器	222		
5.3.1 虚拟内存技术的基本原理	222		

思考题	335	第 8 章 嵌入式系统功耗优化	379
扩展阅读	336	8.1 嵌入式系统功耗优化概述	379
第 7 章 嵌入式系统软件概述	337	8.1.1 嵌入式系统的功耗问题	379
7.1 嵌入式系统的软件框架	337	8.1.2 SoC 芯片级功耗优化	380
7.1.1 嵌入式系统软件所面临的挑战	337	8.1.3 嵌入式系统级功耗优化	384
7.1.2 嵌入式软件的层次框架	338	8.2 SoC 芯片级低功耗设计方法	385
7.2 嵌入式操作系统的基本原理	340	8.2.1 时钟门控	386
7.2.1 嵌入式操作系统简介	340	8.2.2 多电压域技术	388
7.2.2 嵌入式操作系统的内核	341	8.2.3 电源门控技术	389
7.2.3 任务管理与调度	342	8.2.4* 动态电压频率调节和自适应 调节	391
7.2.4 任务间通信	348	案例: SoC 芯片低功耗设计	396
7.2.5 中断管理	350	8.3 嵌入式系统级低功耗设计方法	399
7.3* Android 操作系统简介	356	8.3.1 嵌入式系统级功耗优化技术 介绍	399
7.3.1 Android 操作系统的层次	356	8.3.2 动态电源管理 DPM	400
7.3.2 Android 虚拟机	358	8.3.3 动态电压调节 DVS	402
7.3.3 Android 的任务间通信机制	366	8.3.4 动态电压频率调节 DVFS	404
7.3.4 Android 的安全机制	371	案例: 整机系统级低功耗设计	406
案例: 基于 SEP4020 的 EPOS 软件 平台设计	375	思考题	411
思考题	378	扩展阅读	411
扩展阅读	378		

第1章 嵌入式系统概况

1.1 什么是嵌入式系统

关于嵌入式系统的定义可以说是众说纷纭，IEEE 给出的定义是：“嵌入式系统是用来控制、监控、或者辅助操作机器、装置、工厂等大规模系统的设备。”而维基百科给出的定义则是：“所谓嵌入式系统是指完全嵌入受控器件内部，为特定应用而设计的专用计算机系统。”国内学术界和工业界普遍接受的定义则是：“嵌入式系统是指以应用为中心，以计算机技术为基础，软件硬件可剪裁，适应应用系统对功能、可靠性、成本、体积、功耗严格要求的专用计算机系统。”

总而言之，与巨型机、服务器、工作站和 PC 等通用计算平台不同，嵌入式系统是为特定应用设计的专用计算机系统。所谓通用计算平台，是指计算机的功能主要取决于所运行的软件系统，不同的软件决定了通用计算机系统的功能，比如在服务器上运行 Web 服务器，那么这台服务器的主要功能就是作为网页服务器，而如果运行的是打印服务，那么该服务器的主要功能就是打印服务器。嵌入式设备则通常具有一个非常特定的应用功能，比如即使在今天智能手机的功能已经非常强大，可以集成大量的应用，但是其作为电话的功能却必须是首先实现的。所以从最广义的角度上来看，只要是专用计算机系统都可以称为是嵌入式系统，甚至有种开玩笑似的定义，所谓嵌入式系统就是 PC 取反，所有非 PC 类的计算机系统都是嵌入式系统。甚至 PC 也是由嵌入式系统构成的，比如键盘控制器、硬盘的控制器构成的专用控制系统等等。

从嵌入式系统的定义我们可以知道嵌入式的应用其实无所不在，现代生活每天都在和嵌入式系统打交道。据 ARM 公司的预测，2013 年全球出货的消费电子类产品将达到 24 亿部（其中手机 14 亿部），而围绕工业控制、汽车电子、网络通信等传统嵌入式设备应用，2013 年全球出货的嵌入式微处理器将达到 110 亿颗。可以不夸张地说，嵌入式应用和相关技术支撑了全球电子信息类产业的大半壁江山。

与通用计算平台不同，嵌入式应用往往更加强调系统的高性能、低成本、低功耗以及实时性、可靠性等设计因素。尤其是对于消费电子这类面广量大，强调用户体验，并且采用电池供电的设备而言，系统的性能、成本以及功耗是必须考虑的优化目标。然而，高性能往往意味着高成本和高功耗，如何在性能、成本与功耗间找到折中是这类产品设计过程中必须面对的挑战。本书也试图重点介绍这三个设计维度的考量、优化与折中。

值得一提的是，传统上归入嵌入式系统设备的移动电话等消费电子类产品，由于采用更加强大的处理器以及相关硬件和操作系统，在这些设备上集成的功能也越来越丰富，比如智能手机和平板电脑已经可以实现上网浏览、处理邮件、办公软件、游戏、高清视频、高保真音乐等功能，使得这类产品已经越来越具备通用计算平台的特征，也就是设备的功能取决于其上所安装的软件系统。在这类产品上嵌入式专用计算平台与通用计算平台的界限正在变得模糊。作者认为这种通用化的趋势会变得越来越明晰，智能手机等传统嵌入式设备会逐渐变为通用计算平台，这个通用化的过程其实也正是移动互联网兴起和普及的过程，而与此相应的是手机专用外设和众多以手机为终端的软件商（当然，也可能是服务商和内容提供商）的兴起，这个过程将像当年 PC 的兴起一样充满创新、机遇和激情。

1.2 嵌入式系统的应用与分类

从不同的分类标准我们可以对嵌入式系统应用进行不同的分类，本节将从实时性要求和应用领域两个维度对嵌入式应用进行分类，虽然任何分类方法都很难准确地划分纷繁复杂的不同应用和产品形态。另一方面，由于移动互联网和物联网的兴起，这两大领域的应用已经深刻改变了传统嵌入式系统应用的产品形态和商业模式，因此我们将用两小节的篇幅单独介绍。

1.2.1 基于实时性的分类

所谓实时性是指系统运行的正确性不仅仅取决于功能的正确完成，同时还取决于在规定的时间内完成该功能。按照系统对于实时性要求的严格程度，我们可以简单地将嵌入式系统划分为非实时系统、软实时系统和硬实时系统。

非实时系统。在非实时系统中，系统的功能正确性仅仅取决于功能是否正确执行，而与功能执行的时间无关。简单地说，比如我们在手机或平板电脑上打开一个 Word 文档并进行编辑，这个功能的正确性仅和文档是否正确打开、是否能够正常编辑并保存有关，而与打开文档所耗费的时间无关（打开文档耗时仅仅影响用户体验，而不影响功能）。

软实时系统。软实时系统的功能正确性不仅与功能执行有关，而且该功能必须在规定的时间内完成，否则将造成系统的功能不正常或故障，虽然这种不正常或故障并不会引起崩溃性和灾难性的后果。移动电话的语音编解码系统就是一个典型的软实时系统，电话的编解码系统必须在规定的时间内完成语音的采样、编码并在规定的时间内封装成为可传输的通信帧进行传输。如果系统不能在规定的时间内完成此项工作，就有可能造成通话质量的下降或停顿。另外，移动智能终端上的音乐播放软件和视频播放软件都必须在规定的时间内完成音频或视频文件的解码，否则将造成音乐或视频播放的卡顿。

硬实时系统。硬实时系统要求系统必须在规定的时间内完成规定的功能，否则将造成崩溃性的后果。火箭的控制系统可能是硬实时系统的最好例子，如果控制系统不能在规定的时间内完成对各路传感器传回数据的分析并做出控制响应的话，整个火箭系统将可能出现不可逆的灾难后果。硬实时系统的另外一个比较直观的例子是汽车的安全气囊。

关于嵌入式系统的实时性设计与分析是学术界和工业界研究的一个热门领域，随着系统复杂性的不断增加，如何保证任何情况下都能在规定的时间内做出正确的响应是一个巨大的挑战。

1.2.2 基于应用的分类

正如上一节所述，嵌入式系统的应用领域几乎涵盖了所有非 PC 的计算机系统，因此从应用领域对嵌入式系统进行分类是一件困难的事。通常情况下，我们将嵌入式应用领域划分为以下几大类。

消费电子类产品：毫无疑问，消费电子类产品是嵌入式应用领域最为面大量广、产品形态各异、竞争异常激烈的应用领域。我们可以将这个应用领域进一步细分为个人信息终端产品、办公自动化产品和家用电器类产品。个人信息终端应用包括手机、平板电脑、数码相机、数码摄像机、移动媒体播放器、个人游戏终端等。办公自动化类产品包括打印机、复印机、传真机等。家用电器包括电视机（含互联网电视）、家庭影院系统、机顶盒、冰箱、洗衣机等。

网络通信类产品：网络通信类产品构建了整个信息网络的基础，主要包括交换机、接入设备、路由器、防火墙、VPN 设备等等。

汽车电子类产品：汽车电子领域是嵌入式系统的传统应用领域，主要包括汽车的引擎控制系统、安全系统（防抱死系统、安全气囊）、车载导航系统和娱乐系统等等。由于车载环境比较恶劣（温度、

震动、灰尘等), 车载系统往往对设备的高可靠性、稳定性有着严格的要求。

工业控制类产品: 工控类应用领域也是一个非常宽泛的应用领域, 主要包括工控 PC、程控机床、智能仪表、生产线控制等等。另外, 我们有时也将交互式终端类产品归入工业控制类产品, 这类产品包括各种类型的非 PC 类的网络终端, 比如税控收款机、POS 机具、各种信息查询终端等等。

医疗电子类产品: 医疗电子类产品包括传统的医疗设备和医疗信息化所需要的各类设备。前者包括大型的 CT 机、核磁共振扫描, 也包括小型的生命体征监护仪、呼吸机、血压计等等。医疗信息化类产品涉及的面也非常广泛, 包括药品物流所需要的各种查询终端、病人住院信息查询终端、医生的电子病历等等。

军工及航天类产品: 军工和航天类产品通常情况下是作为武器系统或航天器系统的相关控制系统、导航系统等等, 涉及的面也非常广泛, 通常情况下这类产品都是硬实时系统, 而且对于系统的稳定性、健壮性有着非常高的要求。

1.2.3 移动互联网

移动互联网是一个庞大的系统, 包括移动智能终端(又可以分为智能手机、平板电脑等设备)、互联网络、网络服务(社交网络、游戏、及时通讯)等内容。传统上, 移动智能终端被划入嵌入式设备领域, 因为早期的手机采用嵌入式微处理器, 运行嵌入式操作系统, 其应用也比较单纯。但是现代的智能手机已经配备了功能强大的多核微处理器和外设, 并运行开放的手机操作系统, 用户可以根据自己的喜好选择安装不同的手机应用软件, 从这个角度上来说, 智能手机已经越来越像一个通用计算平台, 越来越多的公司和个人可以为主流的智能手机平台编写应用软件, 与此同时越来越多的能够和智能手机协同工作的硬件设备也层出不穷。因此, 部分研究者认为移动互联网终端, 尤其是运行开放操作系统的智能终端已不能被划分为嵌入式系统产品, 而是应该作为一类独立的计算机类型。我们认为, 移动互联网终端依然保持了很多嵌入式系统的特征, 比如相比于桌面系统和服务器系统而言, 移动互联网终端的硬件系统依然需要尽可能采用低成本和低功耗的设计, 这使得虽然目前的移动互联网终端的硬件系统的功能已经越来越强大, 但总体上依然较同时代的桌面系统要低一个层次。也正因为如此, 对于移动互联网终端的底层软件与硬件的适配与优化就显得更为重要。另一方面, 虽然移动互联网终端已经出现向通用计算平台过渡的趋势, 但最基本的专用功能(比如智能手机的电话功能)依然是至关重要的。基于这两点考虑, 本书依然将移动互联网终端在广义上归为嵌入式系统。

随着互联网技术以及移动智能终端技术的发展, 互联网已经渗透到现代社会的方方面面。据统计, 2009 年全球互联网的渗透率达到了全球人口的 23.8%, 而到 2015 年这个数字将达到 50%。作为互联网的终端设备, 2009 年共有 14 亿台电脑和 47 亿部移动电话, 并且越来越多的用户通过移动智能终端来访问互联网服务。而按照 ARM 公司的统计, 仅 2013 年一年, 全球出货的消费类移动设备的总出货量将达到 24 亿部, 其中仅移动电话一项就占 14 亿部。对于新一代移动互联网终端的产品特征, ARM 公司 CEO Tudor Brown 在 2012 曾预测至少应该包含以下内容(大部分功能在今天已经成为现实): 强大的无线互联网接入能力, 4~7 英寸的 LCD 显示屏, 支持随时在线, 支持常用的办公软件(比如邮件、字处理、电子表格、PPT 和 PDF 等), 支持高清视频、3D 游戏和 Hifi 音乐, 具有 8 小时以上的电池续航时间。

总的来说, 为了支撑这些产品的设计需求, 作为核心的移动终端嵌入式微处理器需要具备高集成、高性能、低能耗的特征。移动设备往往对产品的体积有着严格的限制, 因此为了在有限的空间内支持如此丰富的功能, 移动终端处理器必须将尽可能多的功能模块集成在一颗芯片中。比如, 除了主 CPU 外, 其还必须集成 GPU(图形处理器)、VPU(视频处理单元)、各类存储器接口(DDR, Nand Flash, SD 卡等)、各类通信模块与接口(3G/4G 移动通信, Wifi, USB OTG, 蓝牙, GPS 等)、高清显示控

制与接口（高清 LCDC，HDMI 接口）。如何将如此多的功能模块集成在一颗芯片，并保证每个功能模块的正常工作，已经成为现代 SoC 设计领域的一个重大挑战。另一方面，在保证处理器高集成度：高性能的同时，设计者还必须保证整个系统的低能耗。移动设备往往采用电池供电，降低系统能耗不仅能够延长电池在充电间隔的使用时间，同时也可以降低由于发热而带来的散热问题（毕竟，谁都不想要自己的智能电话需要配备散热风扇）。遗憾的是，系统对于高集成度与高性能的要求与低能耗的要求往往是一个矛盾，如何解决“既要马儿跑，又要马儿不吃草”的问题，成为移动智能终端处理器设计的另一个设计挑战。

1.2.4 物联网

物联网（The Internet of Things）是“万物沟通”的、具有全面感知、可靠传送、智能处理特征的连接物理世界的网络，实现了任何时间、任何地点及任何物体的连结。Kevin Ashton 最早在 1999 年提出了这个术语，虽然当时这个术语更多地用于供应链管理这个领域。今天，物联网可以帮助实现人类社会与物理世界的有机结合，使人类可以以更加精细和动态的方式管理生产和生活，从而提高整个社会的信息化能力。如果说传统的互联网是将全世界的计算机连接在一起的网络，而移动互联网则进一步将人们获取信息、处理信息、发布信息的终端从桌面计算机延伸到移动设备（包括智能手机、平板电脑、可穿戴设备等）的话，物联网则是将信息的获取和处理进一步延伸到几乎所有的物理设备（被称为智能对象，Smart Objects）。物联网的诞生将人类信息网络的神经末梢延伸到了几乎所有领域，包括农业、工业、交通、环境监测与保护、远程医疗、智能家居等。

物联网并不是独立于传统互联网的一个新兴网络，从某种意义上来说它实际构建在传统互联网之上。一般来说我们将物联网分为 3 层：第一层是物理感知层；第二层是网络传输层；第三层是应用层（见图 1-1）。物理感知层是物联网的最底层，负责采集物理世界的相关信息，并对来自上层应用的命令进行响应。采集物理世界的相关信息需要各种传感器来完成，比如温度传感器、光照传感器、摄像头、RFID 等等；为了响应上层应用发出的命令，有时还必须在该层配备相应的执行器（Actuators）。因此，物理感知层通常需要通过无线传感器网络（WSN，Wireless Sensor Network）或者传感器/执行器网络（SANET，Sensor/Actor Networks）来实现。可以说，物理感知层实际上是物联网的最本质特征之一，因为在此之前互联网的终端以桌面计算机和移动智能终端为主，而物联网的物理感知层将各种智能对象连接到互联网。所谓智能对象，可以将其定义为：

- 具有物理特性（大小、形状等）的实体；
- 具备最基本的通信功能，比如可以被其他通信实体发现，并可接受输入的信息和发出相应的回应；
- 拥有唯一的 ID 标识；
- 拥有至少一个名字和一个地址，名字用于人类识别，而地址用于机器间的通信；
- 具备一定的信息处理能力；
- 可以感知物理世界的相关信息（Sensor），或者可以对物理世界施加影响（Actuator）。

一个无线传感器网络通常由大量随机部署的传感器节点设备和一个或少数几个作为连接互联网的网关的汇总节点（Sink）构成。传感器节点只具有非常受限的计算能力、通信能力和电池容量。在网络拓扑结构上，由于受限于传感器节点的电池容量、处理能力以及部署方式，无线传感器网络通常采用 ad hoc^①网络的形式，也就是不需要预先布设的交换节点和路由节点，所有的通信节点在网络中地位相等，每个节点既要发送自己的信息也要承担转发来自其他节点信息的功能（Sink 节点除外，因为该类节点担负着将来自整个传感器网络的信息汇总传输到互联网的功能，因此该节点具有网关的作用）。

① “ad hoc”这个词来自拉丁语，意为“for this”。

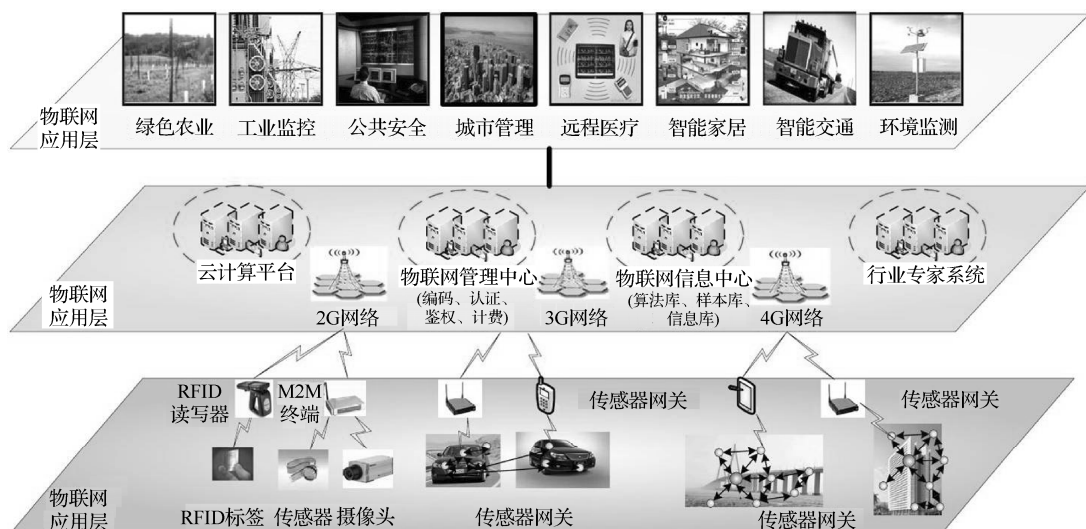


图 1-1 物联网

物联网的第二层是网络传输层。事实上，这里的网络层就是指传统的互联网，借助感知层的网关节点，物联网将底层采集的物理世界信息汇入互联网（也就是运行 TCP/IP 协议的全球网络）。互联网为这些信息提供传输通道和安全控制的同时，也为物联网应用提供了云存储平台和云计算平台。

物联网的最上层是应用层。如果说感知层和网络层提供机制保障的话，应用层才是真正的策略层。比如，微型传感器节点布置在农田当中，动态自组织成网络，采集农田作物的实时信息，如温度、光照度、土壤酸碱度等，并将采集的数据通过无线传感器网络返回到监控中心，监控中心将参照农作物种植的最优化环境数据，采取相应举措，如启动灌溉设备等。再如，如今的工业厂房规模日益增大，厂房间等的实时环境成为影响安全生产和产品质量的重要因素。单靠人力去监测厂房设备及环境信息会耗费大量的资源。如果在广大的厂房空间中布置好一定数量的传感器节点，这些节点就能组成网络实时采集设备及环境信息，如温度、压力、烟雾浓度、有害气体含量等，并将信息实时反馈到监控中心。

从系统的层面来看，物联网应该具备以下特征：

- 支持异构设备。物联网中包含了大量的异构智能对象，这些对象在计算能力和通信能力上都具有很大的不同。为了管理这些异构设备物联网，应该在体系结构和协议两个层面都提供支持；
- 通过近距离无线通信技术实现无所不在的数据交换；
- 低功耗解决方案。由于大量的传感器接点是通过电池或者其他非传统电源方式（比如被动 RFID 采用感应的方式获取能量）供电的，所以低功耗技术和能量获取技术对于物联网具有举足轻重的作用；
- 位置与跟踪能力。通过短距离无线通信技术，物联网中的通信节点可以被定位（甚至跟踪移动轨迹），这个特性对于物流和产品生命周期管理等应用（这类应用已广泛采用 RFID 技术）具有非常重要的作用；
- 自组织能力。一方面由于传感器节点布置的随机性，另一方面由于这些节点在计算能力、通信能力和电池容量上的限制，传感器网络通常采用 ad hoc 网络的形式进行组织，这就要求系统中的各个节点能够自组织起网络的通信方式，比如动态的路由选择；
- 内建的安全与隐私保护机制。因为物联网与物理世界的紧密关联，因此必须从系统的最底层保证安全性和隐私。

通过以上分析，我们可以看出在整个物联网系统中真正与嵌入式系统相关的部分主要集中在物理感知层，主要包括嵌入式微控制器及相关软件、短距离无线通信、传感器等，而其他两层则更多地与传统的计算机科学相关。

1.3 嵌入式系统的产业链

嵌入式系统产业涉及现代电子信息产业的方方面面，几乎涵盖了电子信息产业的所有内容，构成了一个纷繁复杂的产业生态环境，形成了错综复杂的产业生态链。从大的方面进行划分，这个生态环境可以分为芯片设计与制造、方案设计与软件、整机设计与生产以及运营与服务这几个方面（如图 1-2 所示）。

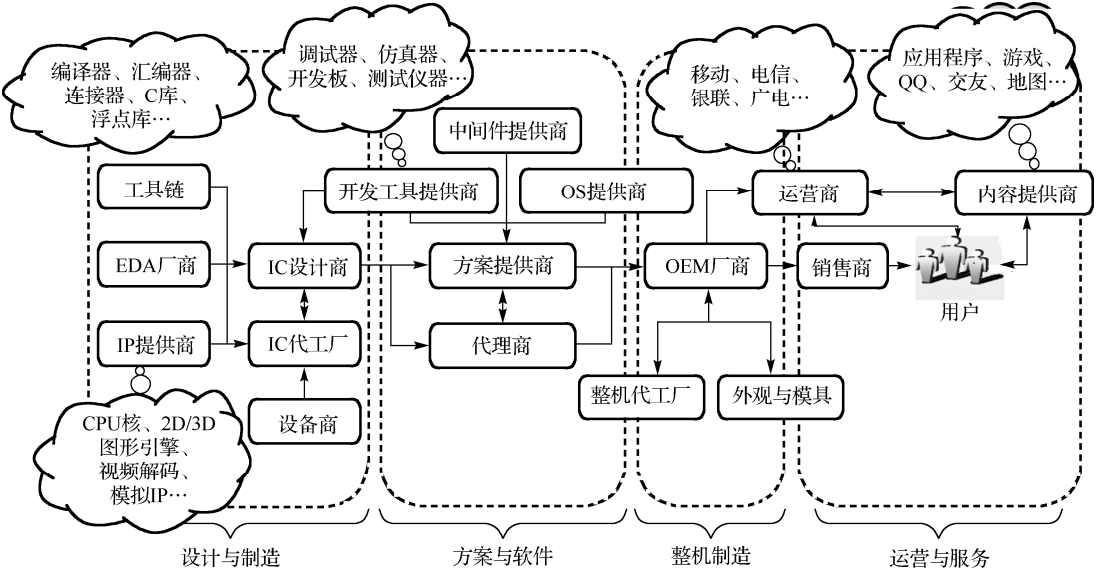


图 1-2 嵌入式系统的产业链

芯片设计与制造又可以分为 IC 设计商（Fabless）、IC 代工厂（包括 Foundry、封测厂等）、IP 提供商（IP Vendor）等。当然国际半导体巨头厂商中有仅负责 IC 设计的纯 Fabless 厂商，比如 Mavell、Broadcom 等；也有整合设计与生产封测于一体的 IDM（集成设备制造商，Integrated Device Manufacture）厂商，比如 Intel、Samsung、ST、NXP、FreeScale 等；还包括仅负责生产或封测的厂商，比如 TSMC（台积电）、SMIC（中芯国际）、日月光等。IP 提供商中有只提供知识产权 IP 核的公司（如 ARM），也有除提供 IP 外同时推出自己芯片的厂商（如高通等公司）。当然，除了前述的几类厂商外，IC 代工厂还需要向相应的设备商购买用于生产、测试、封装的专用设备。IC 设计商也需要在采购所需 IP 外向相应的 EDA（电子设计自动化，Electronic Design Automation）厂商购买用于设计的 EDA 工具，比如 EDA 产业的两大巨头 Synopsys 和 Cadence，当然这些 EDA 公司有时不仅销售 EDA 设计工具，同时也销售自己的 IP 和提供设计服务（Design Service）。

芯片设计和制造出来以后还必须由相应的方案提供商（Design House）基于这些芯片开发相应的方案设计。当然，随着竞争的日益激烈，越来越多的芯片提供商在推出新芯片的同时也会给出相应的非常完善的参考设计（这部分工作有时由芯片提供商自己提供，有时也会委托第三方的方案商进行设计）。另外，通常情况下，芯片提供商会委托芯片代理商进行芯片的推广工作，由芯片代理商直接面对

芯片用户并提供必要的技术支持。为了设计需要,方案提供商往往还会采购第三方的相关工具和软件,比如开发工具提供商、操作系统提供商(比如风河公司、微软公司、Mona Vista 公司)和嵌入式软件中间件提供商(比如媒体解码库、地图导航软件、嵌入式数据库)等。

芯片方案商所提供的解决方案最终被应用于某个整机厂商(OEM 商)的整机产品设计,通常情况下现有的 OEM 厂商仅负责产品的策划、外观设计(最后产品的模具设计一般委托给第三方的模具设计厂商)、功能定义、品牌包装、市场推广等工作,具体的整机产品生产都外包给专门的整机代工厂负责代工,OEM 厂商可能只负责整机的组装和测试。

最终的整机产品将通过相应的销售商渠道推向市场。当然,随着移动互联网的兴起,越来越多的嵌入式整机产品(尤其是消费类电子产品)还需要订购通信运营商(比如中国移动、中国电信、中国联通)和内容服务商(比如腾讯、新浪、京东等)所提供的通信服务和内容服务。

在整个嵌入式系统的产业链分工中,一方面各个厂商的定位与分工越来越细、越来越专业,另一方面产业链上的分工也在出现逐渐融合的趋势。比如,原来专门研发嵌入式 CPU IP 的 ARM 公司越来越关注最终的用户体验,并与众多整机厂商进行合作以保持其 CPU 的架构创新能够满足最终用户的使用体验。再比如很多传统的整机厂商也开始涉足运营和内容服务,并采用全新的营销模式,这其中的代表包括国产小米手机。另外,随着芯片提供商开始为 OEM 厂商提供完整的整机解决方案(Turn Key 方案),进一步压缩了传统方案提供商的市场空间。在这种融合的趋势中做得最彻底的可能是苹果公司了,为了给用户提供了最好的用户体验,苹果的 iPhone、iPad 产品不仅采用苹果自主设计的处理器芯片,自己设计所有的硬件系统,采用苹果自己的 iOS 操作系统,苹果还通过 AppStore 和 iTunes 等网络平台为用户提供应用软件销售、数码音乐销售、电子书销售等在线服务(当然,苹果将产业链中利润较薄的整机生产环节外包给了富士康这样的整机代工厂)。

1.4 嵌入式系统的知识体系

嵌入式系统的本质是专用计算机系统,因此从知识体系上来看嵌入式系统相关技术的知识体系与计算机和电子信息类专业的内容是一致的。就一个具体的应用来看,在应用和物理世界之间存在着巨大差距,为了解决应用的需要,我们需要将应用的问题划分成若干个层次进行解决。一般而言,这些层次包括:器件、电路、寄存器传输级、微架构、指令架构、操作系统、编程语言、算法和最终的应用(如图 1-3 所示)。

所有的嵌入式系统最终都是由数量庞大的电子器件组成的,这些器件最主要的是以 CMOS 工艺为代表的 MOS 晶体管,在器件层面需要研究和学习的内容包括器件原理、器件结构、器件的制造工艺等内容。

器件组成了电路,电路是实现基本逻辑功能的基本单元,在电路层面需要学习的主要知识包括电路分析、电路与系统、数字电路和模拟电路等。

所谓寄存器传输级(RTL, Register Transfer Level)是指以寄存器或门电路为单位对整个电路系统的描述。寄存器传输级进一步对一个计算机系统的逻辑电路进行了抽象,使得设计人员可以以逻辑门和触发器为单位对系统进行设计,在现代数字集成电路的设计过程中,设计人员通常采用硬件描述语言(比如 VHDL 或 Verilog 语言)在寄存器传输级来描述数字电路的逻辑并进行仿真,在基本功能和时序正确后再通过综合(Synthesis)将其转化成为逻辑门和触发器的网表(Schematic,可以理解为基于逻辑门和触发器的电路图)。

微架构是指对寄存器传输级的进一步抽象,如果说寄存器传输级是对电路的抽象的话,微架构实际上已经是电路功能模组的进一步抽象。比如,CPU 内部的流水线设计、分支预测机制的实现、乱序执行机制和指令多发射机制等等。微架构实质上就是 CPU 或其他计算引擎的具体实现。

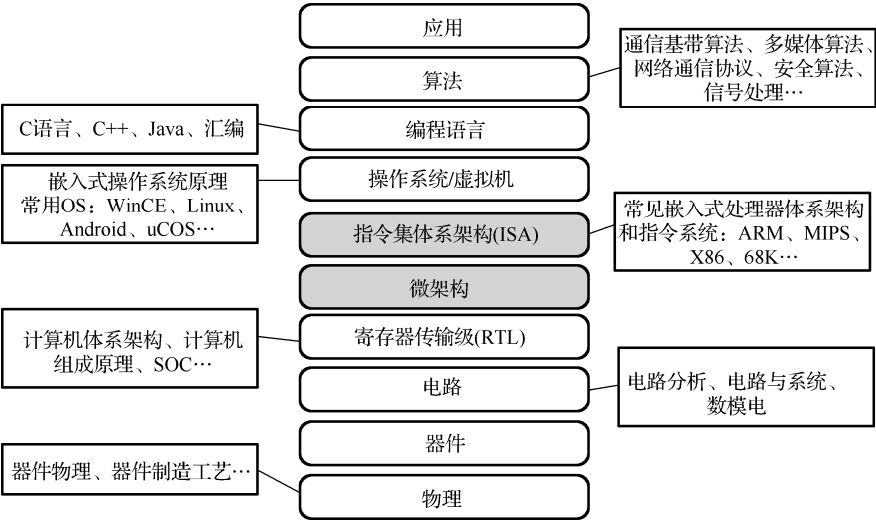


图 1-3 应用的层次

指令集体系架构（ISA，Instruction Set Architecture）是软件和硬件的分水岭，一般而言指令集体系架构之上的层次属于软件层面，指令集体系架构之下则属于硬件层面。指令集体系架构定义了某款 CPU 的指令集，而一款 CPU 是否能够满足某类特定应用的需求在很大程度上取决于指令集的设计。一般而言，指令集系统的设计分为两个流派，一是 CISC 架构，一是 RISC 架构。CISC 架构追求指令集的功能强大，让更多的复杂功能可以通过单条指令完成。但是由于 CISC 架构需要大量的额外硬件去实现使用频度非常低的部分指令，使得其成本、功耗的代价相对比较高，因此从 20 世纪 80 年代开始出现了 RISC（精简指令集）架构，该架构用相对简单得多的硬件实现功能简单的指令集，通过提高主频来弥补指令功能的欠缺。由于 RISC 架构在能效比方面能够取得更好的结果，因此现代嵌入式微处理器设计中采用 RISC 设计思想的实现占据了主流（即使像 Intel 这样传统的 CISC 处理器在其微架构设计上也大量借鉴了 RISC 的设计思想）。我们将在第 4 章详细讨论处理器的指令集体系架构。

指令集体系架构之上的层次属于软件层面，主要包括操作系统、编程语言、算法和应用。嵌入式操作系统已经成为现代嵌入式系统设计不可或缺的软件平台，与传统的桌面操作系统主要作为用户的使用平台不同，嵌入式操作系统则主要是作为设计人员的开发平台。目前常见的嵌入式操作系统主要包括：嵌入式 Linux、微软的 Windows CE 系列、谷歌的 Android、风河的 VxWorks、开源的 uCOS、ThreadX 等。早期的嵌入式编程语言主要是 C 语言和汇编语言，但随着嵌入式操作系统的功能越来越强大，嵌入式软件的规模和复杂度越来越大，C++、Java 等面向对象语言，甚至 XML、Python 等脚本语言也得到了越来越广泛的应用。算法是系统完成一个实际功能所需要采用的方法，在嵌入式系统中所采用的算法与系统的功能有关。比如，对于个人移动互联网终端类（如智能手机）的应用，系统必须包含数字语音处理与压缩算法、移动通信基带算法、多媒体编解码算法（含音频、视频、图像等）、安全认证算法、网络通信协议等内容。事实上，上述任何一个算法所涉及的内容都是一个专门的学科方向所研究的内容。

从上面的分析可以看出，嵌入式系统所涉及的知识几乎涵盖了整个电子信息类专业的所有内容。完全掌握这些知识，尤其是仅仅通过一门课程、一本教材的学习，如果说不是不可能的话也是极其困难的任务。好在对于大多数读者而言，至少在前期的学习实践过程中或多或少对于这些层次的相关内容进行了专门的学习，而本书的目的就是在读者具备相关知识的基础上，给出一个完整的嵌入式系统的框架和脉络，为读者的后续学习奠定基础。本书的主要内容将重点关注指令集体系架构、微架构和

寄存器传输级三个层次，使读者站在 SoC 设计者的视角学习嵌入式系统的构成。至于电路、器件和物理三个层次的内容，电子类专业的读者应该在先修课程中学习。编程语言、操作系统的相关内容本书会简单介绍，这部分的详细内容将在后续课程中进一步学习（对于计算机专业类的读者，这部分内容其实是先修课程的内容）。至于算法层面，正如前面所说，任何一个具体的算法实际上都是一门独立的学问，需要读者根据未来工作的需要有选择地进一步学习。

1.5 案例：MP3 播放器

本节将以一个 MP3 音乐播放器作为案例来讨论一个简单嵌入式系统的工作过程。我们假设该 MP3 播放器通过键盘进行操作，并且有一个彩色液晶屏作为显示输出。用户的音乐文件（MP3 文件）存放在系统的 Flash 存储器中，系统的 RAM 存储器采用外挂 DRAM 实现，系统的音频 Codec 通过 IIS 或者 AC97 接口与主芯片连接，解码方式为软解码，系统架构如图 1-4 所示。

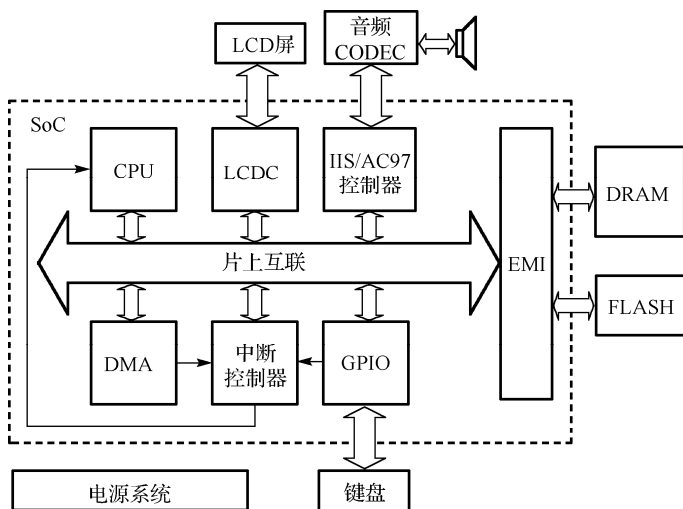


图 1-4 MP3 播放器系统架构

当然，在一个真实的 MP3 播放器实现中，系统可能高度集成，比如 Codec 和电源系统是集成在主芯片 SoC 中的，系统采用专用硬件解码而不是软解（这样可以获得更高的效率和更低的功耗），甚至键盘控制也专门集成了键盘控制器，而不需要通过 GPIO 进行管理。不过为了讨论的方便，我们采用图 1-4 的架构。

假设用户通过键盘操作选择了一首歌曲，并选择播放该文件，系统将如何完成这个任务呢？我们将按照系统处理的顺序介绍这个过程。

① 用户键盘操作最终体现为 GPIO 口上的一个中断信号，GPIO 向中断控制器提出中断请求，在中断控制器完成中断仲裁后，由中断控制器向 CPU 发出中断请求，CPU 在响应这个外部中断后，通过查询中断控制器中的相应寄存器可以获知本次中断由 GPIO 发起，系统进入 GPIO 的中断处理代码，发起对 GPIO 组成的键盘系统进行扫描以获得具体的键值，并以信号、事件或消息的形式通知操作系统。至此完成本次中断服务程序。

② 操作系统在获得键盘消息后，由内核调度相应的任务（线程）对该消息进行处理。通常这个任务是负责 MP3 音乐播放的主线程（将负责所有的用户操作响应、界面显示等）。主线程根据用户选择的文件，将通过文件系统读取将要播放的 MP3 文件。

③ 文件系统在接收到主线程给出的文件读取命令后将通过底层驱动程序访问物理的存储介质，通常情况下驱动程序将配置 DMA 控制器，启动其中的某个传输通道，DMA 在启动后将按照配置内容从外部存储器接口（EMI）读取 Flash 存储器中的文件内容到 DMA 通道的内部 FIFO，并将 FIFO 中的内容通过 EMI 传输到相应的 DRAM 缓冲区。由于受限于内部 FIFO 的大小限制，每次 FIFO 传输完成后，DMA 控制器将发起中断，通知 CPU 重新配置新的 DMA 参数继续完成后续的数据传输，直到文件的内容全部（或部分）传输到 DRAM 中的媒体数据缓冲区。

④ 当 DRAM 中的媒体数据缓冲区就绪后，操作系统将启动负责解码的任务（解码线程），解码线程由 CPU 从 DRAM 的缓冲区取出需要解码的数据并进行解码计算，并将解码后的数据写入 DRAM 的输出缓冲区。

⑤ 当 DRAM 中的输出缓冲区有数据后，操作系统将启动播放任务（播放线程），播放线程将配置 DMA 控制器的另一个数据传输通道，该通道从 DRAM 输出缓冲区中取出数据存放在 DMA 的 FIFO 中，并将 FIFO 中的数据传输给 IIS 控制器，该控制器将按照 IIS 标准的规定将解码后的音频数据传输给片外的 Codec 芯片，Codec 芯片将输出的 PCM 数据通过 D/A 转换为音频信号，并最终播放出来。这个过程和步骤③类似的是，DMA 控制器将不断地在一次 FIFO 数据空的情况下发起中断，CPU 重配 DMA 继续后续的数据传输。

⑥ 播放线程在播放的过程中将不断地通过操作系统提供的任务间通信机制，不断通知主线程播放的相关信息，主线程根据这些信息不断修改显示界面的内容（比如时间、歌词等）。主线程配置 LCDC 不断从 DRAM 中的显示缓冲区将数据刷新到 LCD 屏上，更新显示内容。另外，对于电池供电的设备而言，主线程可能还需要负责系统的低功耗控制。

通过上述分析不难发现，在 MP3 播放的过程中至少存在 5 条数据流：从 Flash 到 DRAM 的数据流，CPU 从 DRAM 取数据进行解码并将结果写回 DRAM 的数据流，从 DRAM 缓冲区到 IIS 控制器的数据流，LCDC 不断周期性地从 DRAM 取显示缓冲区数据的数据流，CPU 在这个过程中不断从 DRAM 中取指令的数据流。这 5 条数据流都需要通过 SoC 片上的片上互连网络和外部存储器接口（EMI）进行数据传输，因此 EMI 以及外部存储器的效率以及片上互连网络的效率直接决定了整个系统的效率。

另一点需要说明的是，从软件角度看，我们可以通过 3 个任务（或线程）协同工作的方式来解决 MP3 播放的任务。其中主线程负责维护用户界面和控制，解码线程负责具体的数据解码工作，而播放线程则负责将解码后的数据传输到具体的 IIS 控制器，三个线程间通过操作系统提供的任务间通信机制进行同步。其中解码线程与播放线程属于典型的生产者与消费者关系。当然，不同的实现可以采用不同的方案，甚至在简单系统中通过单任务也可以完成所有上述工作。

综上所述，即使像 MP3 播放器这样简单的系统也是一个由硬件和软件协同工作的系统，当然从产品设计的角度看，一款 MP3 产品的设计将更加复杂，包括产品的策划、市场、外观设计、生产、销售渠道和售后服务等，绝不仅仅是我们提到的技术实现部分的内容。

思考题

1. 在本章的案例中，LCDC 需要不断地从 DRAM 存储器中取出显存中的数据并将这些数据按照一定的协议传输到 LCD 屏，屏上集成的驱动芯片将按照 LCD 的时序将这些数据显示出来，假设屏的分辨率是 800×600 ，每个像素需要采用 24 位表示（RGB 三个分量各占一个字节），请计算每帧 LCD 数据的大小。假设为了使屏幕的显示不出现抖动，LCDC 需要每秒刷新显示 50 帧，请计算 LCDC 需要每秒从 DRAM 中取多少数据？

2. 在本章的案例中，音频数据按照 44.1K 的频率进行采样，每个采样点需要占用 16 位，按照左右两个声道进行采样，请计算 DMA 每秒需要向 IIS 控制器传输多少解码后的数据？这个过程需要占用多少片上互连的带宽？

3. 请说说能耗与功耗的关系？通过降低系统的功耗是否一定能够降低系统的能耗？电池使用时间的长短主要取决于系统的能耗还是系统的功耗？

扩展阅读

- [1] Barry, Peter and Patrick Crowley, Modern embedded computing: designing Connected, Pervasive, Media-Rich Systems[Book], ELSEVIER, 2012.
- [2] John L. Hennessy, David A. Patterson. 计算机体系结构量化研究方法（第 5 版）. 贾洪峰译. 北京：人民邮电出版社，2013.
- [3] Hisa Ando. 支撑处理器的技术. 李剑译. 北京：电子工业出版社，2012.
- [4] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, ImrichChlamtac, Internet of things: Vision, applications and research challenges [J], Ad Hoc Networks 10 (2012) 1497-1516.

第 2 章 嵌入式系统中 SoC 的硬件架构

本章将介绍 SoC 的主要组成，使读者对其技术细节及其在目标系统中发挥的作用有所了解。对于嵌入式系统的软件开发人员来说，特别是系统驱动的开发人员，了解目标系统的相关硬件细节，可以帮助其更好地利用硬件功能，甚至协助硬件开发人员进行硬件设计改进。了解目标系统的硬件细节，不仅要理解系统内各模块的物理连接关系，更重要的是从软件的角度看待各个组成模块，明白各模块是如何通过驱动软件互相作用以实现复杂的系统应用的。

2.1 SoC 硬件架构概述

在介绍 SoC 硬件架构之前，我们首先介绍一下经过多年发展的 PC 硬件架构。图 2-1 是两个时代（Pentium4，Core2 双核/四核）的计算机硬件架构。

图 2-1 中的两个硬件架构图都可以抽象为图 2-2，从图中我们发现 PC 架构是基于北桥和南桥两个桥接芯片的（作者一直纳闷为什么叫北桥和南桥，可能因为上北下南的意思？），作为主处理器的伴侣芯片，北桥芯片为高速设备提供了非常高速的数据通路，除了连接主处理器外，通过前端总线连接系统主存储器，通过显卡总线连接显卡（包括图形处理器和显存）。北桥芯片通过总线连接南桥芯片，南桥芯片扩展出了 PCI 总线，用于连接 PC 系统的其他板卡，通过 SATA 总线连接硬盘系统，有些南桥芯片还包括高速以太网控制器、USB 控制器和音频控制器。总之，PC 系统是通过北桥和南桥芯片连接整个系统的，而整个系统是集成在主板上的。

与 PC 系统不同，嵌入式 SoC 芯片往往将整个系统集成在一个硅片上，包括存储控制器、图形处理器、通信接口等。国内外学术界一般倾向将 SoC 定义为：集成微处理器、模拟 IP 核、数字 IP 核和存储器（或片外存储控制接口）的单一芯片。如果说微处理器是大脑，那么 SoC 就是包括大脑、心脏、眼睛和手的系统。SoC 就是一个微型系统，可以认为是计算机系统的一个子集。

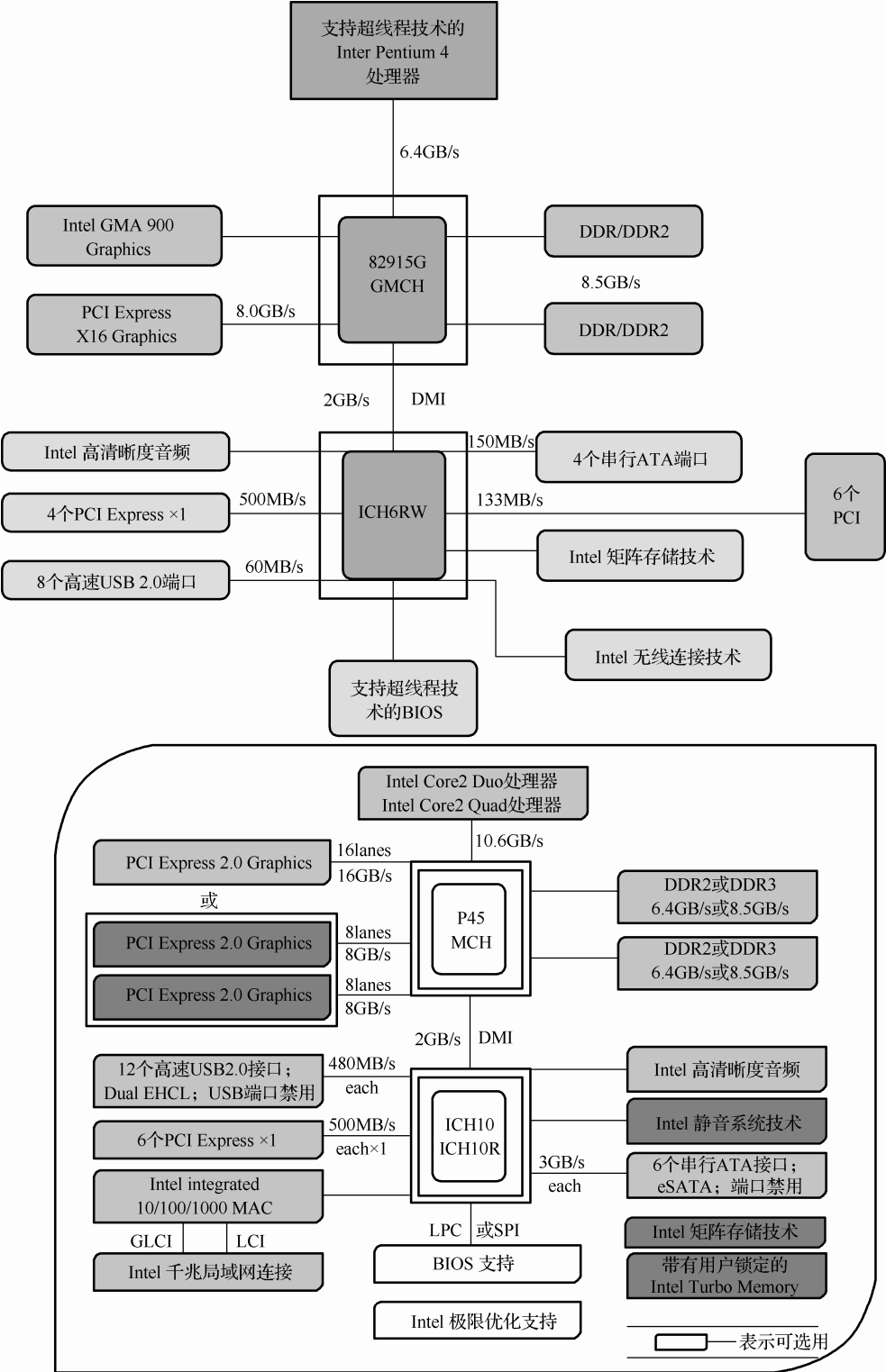
计算机系统存在如下两个概念。

- 体系结构（Architecture）：指对程序员可见的系统属性，如指令集、数据类型、输入/输出机制、内存寻址机制。
- 组织结构（Organization）：指实现结构规范的操作单元及其相互连接，对程序员透明（也就是从程序员的角度看不到这些），如控制信号、模块接口、存储器使用技术等。

SoC 体系结构取决于该 SoC 中所集成 CPU 的体系结构，比如集成 ARM CPU 或者 MIPS CPU，甚至在同一 SoC 中集成多个不同类型的 CPU，对于这种情况我们称其为异构架构（Heterogenous Architecture）。而 SoC 具体的组织结构通常称为 SoC 硬件架构。硬件架构对于 SoC 的功能实现起着至关重要的作用。本节将概述 SoC 硬件架构，介绍 SoC 硬件架构设计的基本规律。

前面所述的使用不同内核的 SoC，其架构基本是一致的，都可以抽象为图 2-3 所示的结构。

与传统的桌面系统（PC）架构不同，现代嵌入式系统 SoC 往往需要将系统的大部分功能模块集成到一个芯片（或一个封装）内，因此将所有这些功能模块互联成为系统的互联架构无疑是 SoC 芯片的内部高速公路，对于 SoC 的性能、功耗有着决定性的作用。为了兼顾高带宽设备与低速设备对于数据吞吐量的不同需求，并最大程度降低系统能耗，现代 SoC 架构中往往采用高速互联架构与低速总线混合的架构。我们将在 2.2 节重点讨论 SoC 的互联架构。



Intel P45 Express Chipset Block Diagram

图 2-1 Pentium4 和 Core2 双核/四核的计算机硬件架构

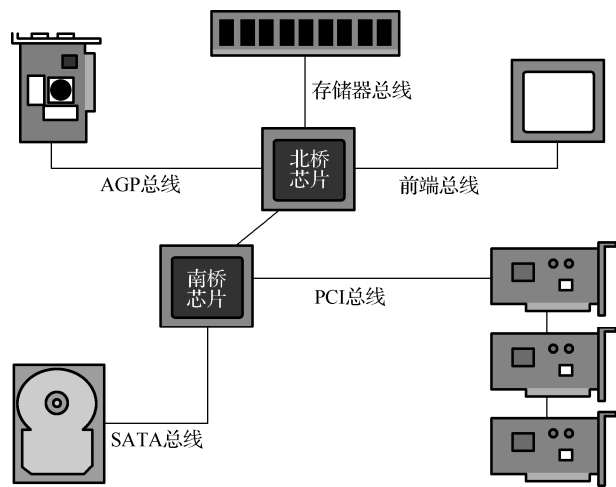


图 2-2 PC 硬件架构抽象

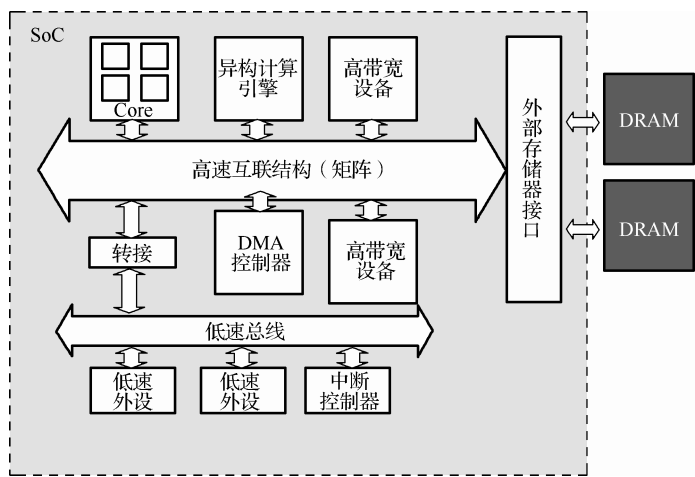


图 2-3 SoC 硬件架构抽象图

SoC 的核心是处理器（CPU）内核，随着现代嵌入式应用的功能复杂性不断提高，SoC 中所集成的处理器内核的性能也越来越强大，以往通常在高性能桌面系统甚至服务器系统中采用的处理器设计技术也不断地被融合到嵌入式 CPU 的设计中，比如多核技术（CMP）、超深流水线技术、指令分支预测、多级高速缓存、乱序执行技术和指令多发射技术以及单指令多数据（SIMD）技术等。但与传统桌面系统以性能优化为主要目标不同（现代桌面系统 CPU 也要考虑成本和功耗因素），嵌入式 CPU 设计需要在性能、成本、功耗之间进行权衡与折中。我们将在第 4 章重点讨论嵌入式 CPU。

一般而言，在 SoC 的高速互联架构中不仅挂接了作为主控核的嵌入式 CPU，还挂接了其他对带宽有较高要求的其他高速设备，比如 LCD 控制器、DMA 控制器等。另外，随着系统（尤其是以智能手机为代表的消费电子类应用）对于图形、图像、视频、高保真音频等多媒体性能的要求不断提高，此类 SoC 中往往还在高速互联架构上集成了专门用于处理媒体数据的计算引擎（比如图形处理单元 GPU、视频处理单元 VPU 以及面向通用计算的可重构计算架构）。在高速互联架构上的所有设备都对存储器带宽有着较高的要求。我们将在 4.4 节介绍异构计算引擎。

如果说高速互联架构是 SoC 芯片内部的高速公路,那么存储子系统无疑就是 SoC 芯片的仓库。冯·诺依曼架构的本质特征就是程序存储的思想,所有需要处理的数据和处理这些数据的指令以及处理完成的结果最终都是存放到存储器中的。因此,SoC 的存储子系统也是整个 SoC 性能、成本、功耗的瓶颈。我们将在第 5 章专门讨论存储子系统的问题。

SoC 中的低速设备通常包括定时器 (Timer)、串口 (UART)、通用 IO (GPIO)、音频接口 (IIS 或 AC97)、其他通信口 (SPI、IIC 等)。这其中一个重要的模块是中断控制器。中断控制器负责接收所有 SoC 芯片模块的中断请求,并按照预订的优先级和屏蔽码决定最终选择哪个中断源向 CPU 提起中断。我们将在 2.7 节及第 6 章详细介绍这些低速设备。

2.2 互联结构

在芯片集成度和设计复杂度越来越高的今天,通过 IP 核重用将整个系统集成在一块芯片上已经成为主流的设计方法。SoC 将处理器内核 (Processor Cores, 如 CPU 和 DSP)、存储器、专用功能/算法模块、外设接口等多个模块通过互联结构连接起来实现系统的功能。由于 SoC 中使用了越来越多的内核和其他模块,它们之间的通信需求也越来越高。为满足这些通信需求,各种片上互联机制得到研究和应用,而它们在 SoC 设计中的作用也越来越重要。常见的互联架构包括共享总线 (Bus)、点到点连接 (P2P) 和片上网络 (NoC),下面我们将对这 3 种互联方式进行介绍。

2.2.1 常见互联结构分类

1. 共享总线 (Bus)

如图 2-4 所示,在总线结构上,传输 (或者交易, Transaction) 都由主模块 (Master) 发起,从模块 (Slave) 负责响应。当主模块发起总线交易请求 (Request),并得到总线仲裁器 (Arbiter) 的许可 (Grant) 后,主模块占用总线,进行对某个从模块的数据传输。当一次传输完成后,主模块释放总线,其他的主模块可以通过仲裁器竞争下一轮的总线传输。

总线的基础结构是由仲裁器、主从模块多路选择器、译码器等模块所组成的总线控制部件。

① 仲裁器的功能是当总线支持多个主模块时由总线仲裁器来仲裁占用总线的传输模块。仲裁器监视主设备发出的总线请求,根据内部设定的仲裁算法进行仲裁,并给出相应的控制信号,保证在任何时候只有一个主设备可以进行数据传输。

② 译码器则负责地址的译码,它将主设备发出的地址 (或地址的一部分) 译码为一组片选信号,这些片选信号将选通此次传输的从设备。

③ 多路选择器使整个总线结构互联起来,它把所需的控制信号和数据路由到相应的目的设备。为完成相应的路由功能,它分为主设备到从设备多路选择器和从设备到主设备多路选择器。

共享总线在芯片设计中使用最为广泛,技术也很成熟。但对于通信要求很高的 SoC 来说,共享总线结构很容易成为性能的瓶颈。因为当总线通信非常频繁时,总线的有效带宽实际上是被多个主设备时分复用的。因此每个主设备的实际带宽将只有总线带宽的 $1/N$,其中 N 为总线上的主设备数。

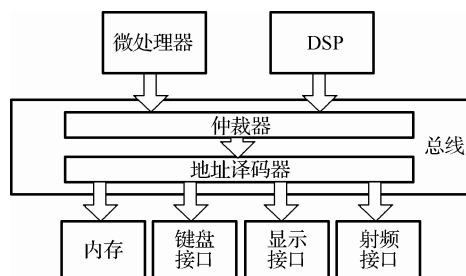


图 2-4 共享总线

2. 点到点连接（P2P, Peer to Peer）

对于点到点结构，主设备与从设备之间存在独立的读写通道，能较好地满足高性能 SoC 设计中大量高速数据传输的要求。如图 2-5 所示，在微处理器进行显示接口操作时，DSP 模块可以同时 对射频接口进行数据传输，从而有效地提升系统性能。

点到点连接的一个实例为 Synopsys 公司推出的互联模块 DW_axi，其结构框图如图 2-6 所示。

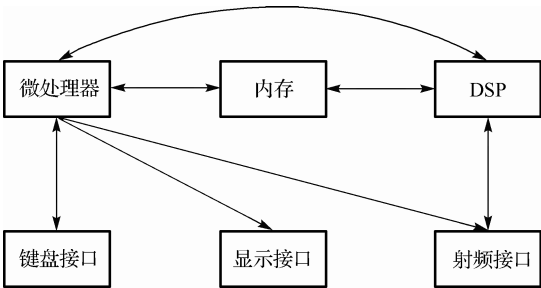


图 2-5 点到点连接

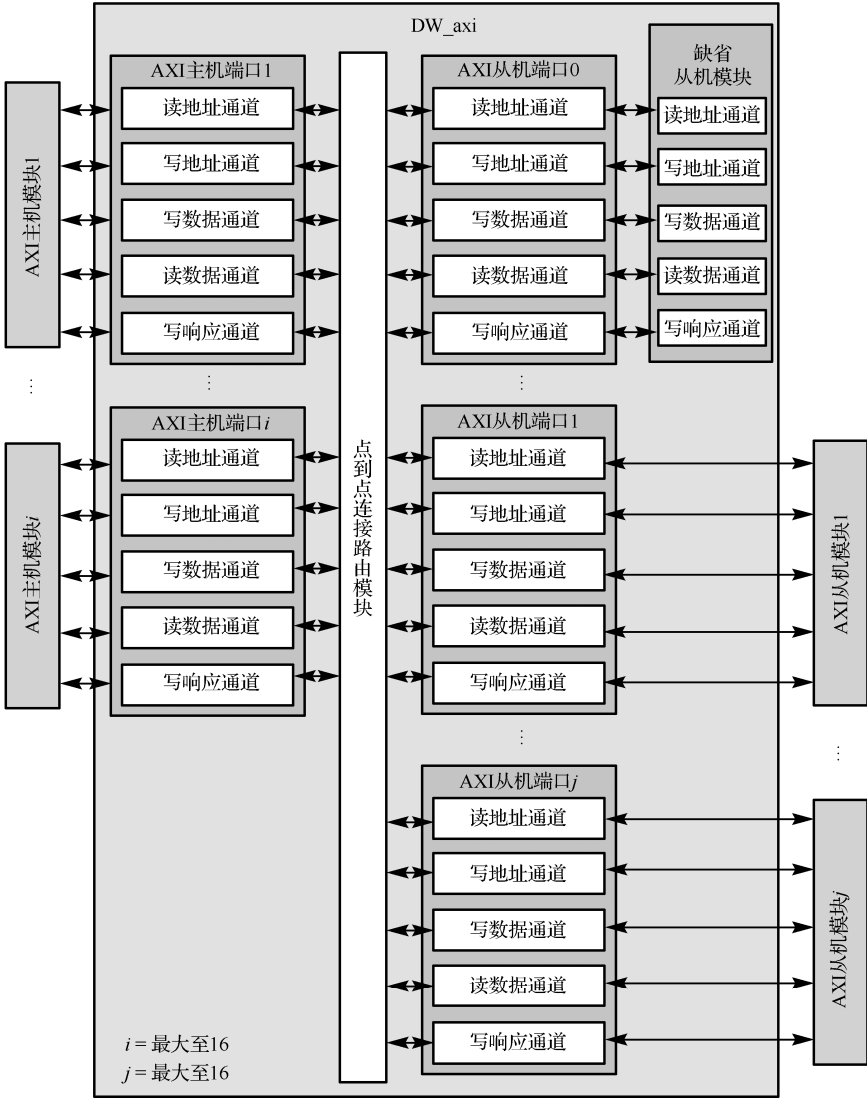


图 2-6 互联模块 DW_axi

图 2-6 中交叉连接路由模块（Crosswire Router）根据主机模块（AXI Master 1~i）的访问需求，连接相应的从机模块（AXI Slave 1~i），形成交叉连接结构。

可以看出点到点连接是基于共享总线结构的一种改进, 类似于多层总线结构。交叉连接可以克服共享总线结构的一些问题, 但它的扩展性受到很大的限制。当设备数量增加的时候, 连线的数量会呈指数增加, 如果要实现全互联, 对于含有 N 个节点的 SoC, 专用数据连接的数量会达到 $N \times (N-1) / 2$, 这会带来两个问题: ①导致芯片面积和布线难度的增大; ②每个设备都需要设计 $(N-1)$ 个通信端口, 显然这会大大增加设备接口的设计难度, 如图 2-7 所示。

3*. 片上网络 (NoC)

目前 SoC 中各功能 IP 核的互联主要是基于总线的通信方式。随着集成度的不断提高, 单一芯片上集成的功能 IP 核不断增多, 传统的总线结构暴露出越来越多的问题, 其中并行通信能力差、全局时钟同步难以实现以及可扩展性差等方面严重制约了 SoC 的发展。为了更好地解决以上问题, 近年来研究者们提出了片上网络 (Network-on-Chip, NoC) 的概念来解决以上问题。基于片上网络的 SoC 结构如图 2-8 所示, 其中 NI 表示网络接口 (Network Interface)。

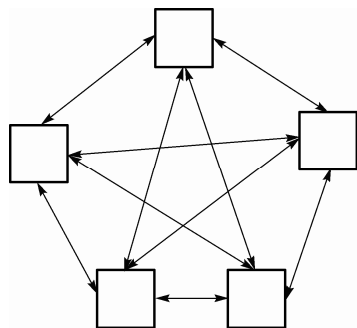


图 2-7 5 个设备节点的全互联结构

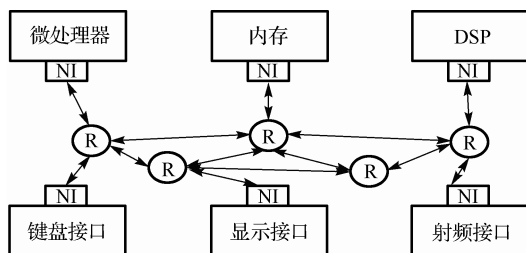


图 2-8 基于片上网络的 SoC 结构

片上网络的概念基于网络通信, 用路由和分组技术来完成通信任务, 以提高系统的并行性和可扩展性, 解决复杂 SoC 的片上通信问题。片上网络采用异步消息传递进行通信 (这一点和传统的总线架构具有很大的不同, 通常总线结构要求主设备和从设备间必须进行同步数据传输, 也就是主设备和从设备之间需要同步时钟), 这样就把整个设计分解成更容易管理的单元, 并可根据应用需求对每一个子系统进行优化。这样一种可扩展的、模块化的结构是系统结构发展的自然选择, 特别是当考虑到物理实现问题时, 情况更是如此。

片上网络自 1999 年首次被提出以来, 由于其高度并行的通信能力、全局异步局部同步 (Globally Asynchronous Locally Synchronous, GALS) 的低功耗特性以及良好的可扩展性, 已经成为目前集成电路业界研究的热点之一。著名咨询公司 Gartner Research 发表的集成电路技术发展规律分析报告形象地描述了每项技术从诞生到成熟的过程, 并将已有的各种技术所处的发展阶段标注在图上, 2011 年的分析报告如图 2-9 所示。

图 2-9 中横坐标表示某一新技术随时间不断提升的成熟度, 纵坐标表示业界对该技术的期待程度, 其中的曲线表明, 每项技术的发展过程均可分为 5 个阶段。

① 一是萌芽期 (Technology Trigger), 又称感知期。人们对新技术产品和概念开始感知, 并且表现出兴趣。

② 二是过热期 (Peak of Inflated Expectations)。人们一拥而上, 纷纷采用这种新技术, 讨论这种新技术。典型的成功案例往往会把人们的这种热情加上把催化剂。

③ 三是低谷期 (Trough of Disillusionment), 又称幻灭期。过度的预期, 严峻的现实, 往往会把人们心里的一把火浇灭。

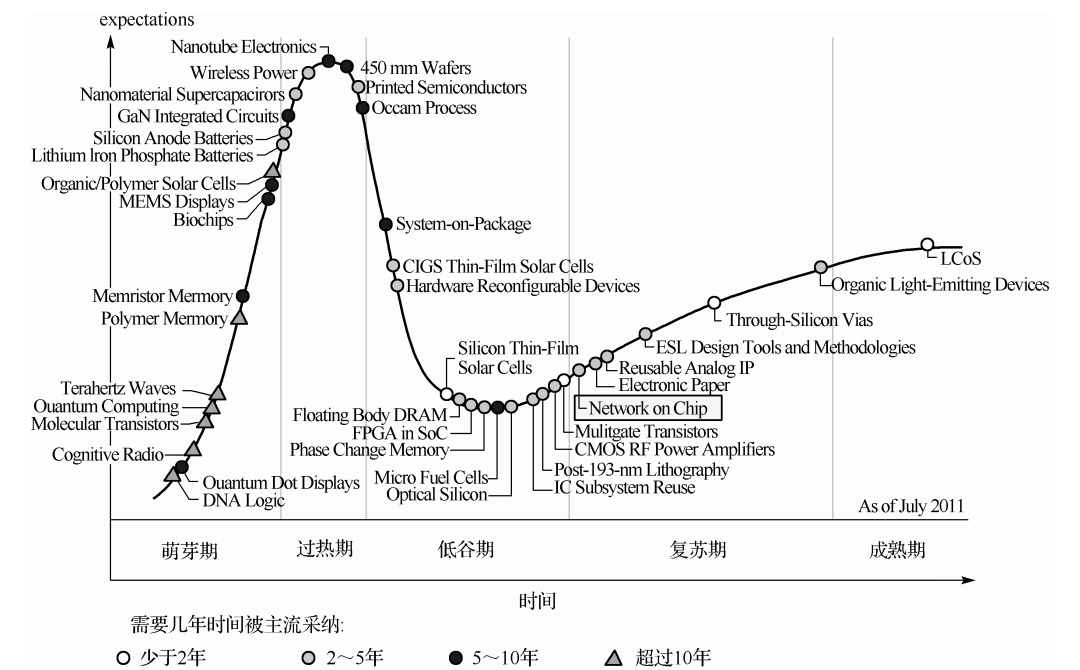


图 2-9 2011 年集成电路发展规律分析报告

④ 四是复苏期（Slope of Enlightenment），又称恢复期。人们开始反思问题，并从实际出发考虑技术的价值，相比之前冷静不少。

⑤ 五是成熟期（Plateau of Productivity），又称高原期。该技术已经成为一种常用技术。

从 Gartner 公司 2011 年的技术成熟度报告中，我们可以看到片上网络现在已经绕过了应用上的瓶颈，开始真正“落地”，逐渐走向成熟。

NoC 是由很多的网络单元（network tiles）通过片上路由进行互联的。网络单元由 IP 核、网络接口组成。网络单元代表处理单元或存储单元，像瓦片一样进行连接，形成拓扑结构。图 2-10 为 NoC 基本结构示意图。

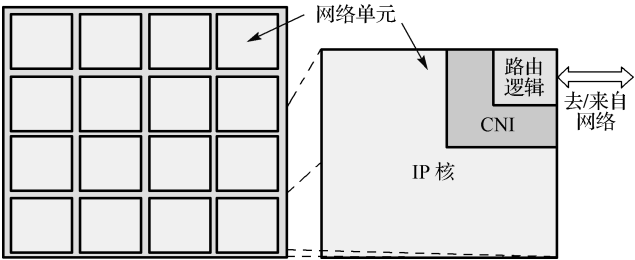


图 2-10 NoC 的体系结构

1) 拓扑结构

NoC 的拓扑结构在很大程度上借鉴了宏观计算机网络，即并行计算机互联网络结构。但是由于 NoC 的特殊性，如有大量可用的互联通信资源可用、核间可以进行大数据量通信、受到片上面积制约、需要规则的网络布局、要有可扩展性等，因此，NoC 对于拓扑结构有着一定的特殊要求。NoC 的拓扑结构定义了网络内节点与链路的布局和互联方式，决定了网络中的路由策略与映射算法，对于网络的时延、吞吐率、面积、容错、功耗等指标有着至关重要的影响。

NoC 的拓扑结构包括规则拓扑与非规则拓扑,前者具有规则的网络参数,具有可复用性,降低了设计时间与成本,适用于通用的对称、同构系统,规则的网络单元拓扑结构有利于进行性能预测设计,而非规则拓扑则需根据特定的应用需求定制,可以面向不同的应用领域。

规则拓扑主要可分为两大类:直接网络拓扑结构与间接网络拓扑结构。

直接网络拓扑结构中,网络内任一路由节点都与 IP 核直接相连。常见的直接网络拓扑结构有网格拓扑(mesh)、花托拓扑(torus)。2D 网格拓扑是最简单的一种结构。如图 2-11 所示,每一个非外围的路由节点都与最近的 4 个节点双向连接,同时也与它相关的 IP 核双向连接。外围节点至少有一个边没有连接。2D 花托拓扑是 2D 网格拓扑的变形,包括外围节点在内的所有节点都与其最近的 4 个节点相连,这意味着一边的节点能够直接把数据传输到另一边。

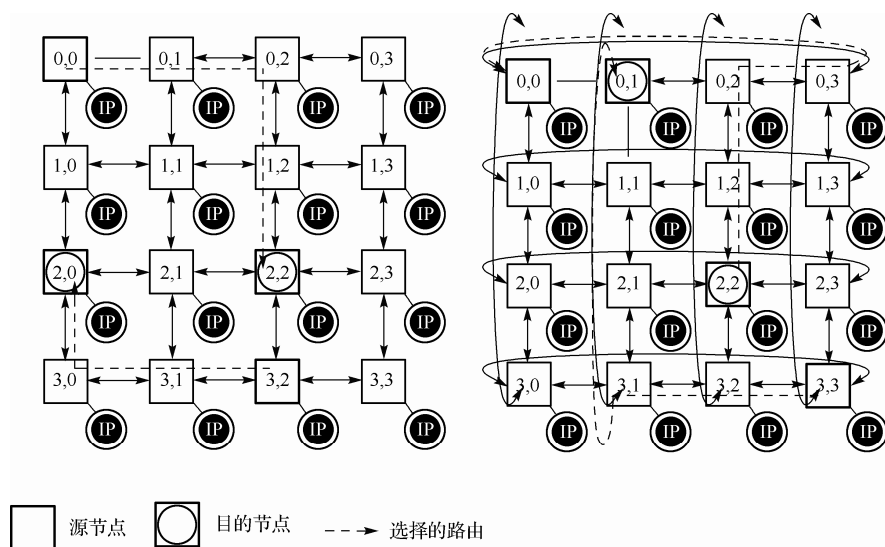


图 2-11 2D 网格拓扑与花托拓扑

间接网络拓扑结构中,网络内一部分路由节点只与其他路由节点相连,与 IP 核并不相连。间接网络可以使得节点设计和网络设计分开进行,提高设计效率。常见的间接网络拓扑结构有蝶形拓扑、树形拓扑,如图 2-12 所示。

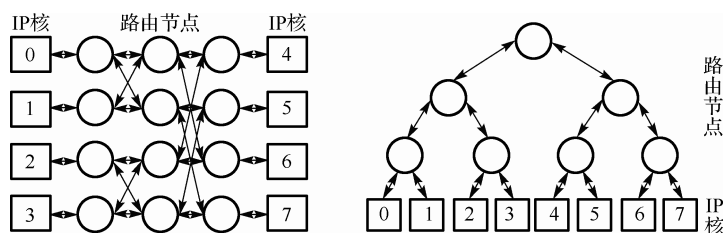


图 2-12 间接网络拓扑结构

2) 路由算法

对于每个消息来说,到目的地的路径一般有多条,路由算法用于决定采用其中的哪一条来传递消息。

路由算法根据选择路径的概率可分为确定性路由与自适应路由。在确定性路由中,选择的路径是预先确定的,只要给定起点与终点,不论网络状态如何选择的路径都是相同的。比较著名的确定性路由算法是维度路由(Dimension Order Routing, DOR),在二维 mesh 网络中,消息先沿 X 维路由再

沿 Y 维路由，在 X 与 Y 方向上逐步靠近，直到目的地为止。在自适应路由中，给定相同起点与终点，会采取不同的消息传递路径，可根据当前的网络流量状况做出相应的决定。自适应路由算法面临的主要问题是死锁问题。死锁发生在数据包因等待另一数据包释放相关资源而不能向前传递，等待序列形成一个环路时，网络即被锁死。避免死锁的基本思想是通过增加某些路由约束，规定不可通过某些路径到达目的地，以防止形成环路。

目前，路由算法主要是针对单播传输的，即从 1 个资源节点到另外 1 个资源节点，对多播传输的支持是未来 NoC 路由算法重要的研究方向，即从 1 个资源节点到多个其他资源节点。

3) 交换策略

交换策略决定网络中的数据包如何进行传递以及随着时间的推进可利用哪些网络资源（比如缓冲区和链路）。片上网络中运用的交换机制主要可以分为两类：面向连接的和无连接的。面向连接的交换机制主要有电路交换，无连接的机制包交换主要有存储转发（Store-and-Forward）、虚切通（Virtual Cut-through）和虫洞交换（Wormhole Switching）。

电路交换需要在通信双方之间成功建立连接通道之后，双方的通信活动才能开始。而包交换则不需要在数据传输之前建立连接通道。

存储转发是最为简单的包交换机制，数据包（Packet）在每个网络节点（Router）被完整缓冲之后才能被转发至下一网络节点。

虚切通交换将数据包更进一步划分为更小的微片（flit），大小通常为若干个比特，并将所有的微片按顺序排好，将所需的路由信息放入第一个微片中（称作头微片，后续微片称为数据微片）。由于路由信息只包含于头微片中，所以路由节点没有必要等整个数据包都接收完以后再进行转发。在无阻塞的情况下，路由节点收到头微片后，从中读取路由信息，然后由路由决策单元负责选路，如果输出通道空闲，则将头微片转发出去，后续微片紧随头微片向前路由，从而大大缩小了包交换的延时。当头微片所请求的输出通道全忙时，头微片就地缓存在中间节点，随后的数据微片也依次前往并缓存在该节点中。如果阻塞的时间足够长，则整个数据包的微片都将存放在该中间节点中，因此像存储转发一样，中间节点也要提供至少一个数据包大小的缓存资源。

虫洞交换是当今片上网络中的主流交换机制。它和虚切通交换的基本思想大体相同，只是二者在发生阻塞时所表现出的行为不同。在虫洞交换中，微片的种类分为头微片、数据微片和尾片，并且允许一个数据包只由一个微片组成。当头微片发生阻塞时，数据包中的所有微片都将停止前进，也就是说发生阻塞时头微片缓存在当前节点，数据微片就地缓存在其后的若干个中间节点中。每个路由节点只需提供一个微片大小的缓存资源。在无竞争的情况下，虫洞交换和虚切通交换的时延性能基本相同，并且可以通过在每个路由节点适当增加缓存数量来进一步提高吞吐量。优良的时延性能、较小的缓存要求以及大吞吐量是虫洞交换最突出的优点。虫洞交换也有自身的一些缺点，当发生阻塞时，虫洞交换中的各个微片将会就地阻塞，从而使信道相关性扩展到了多个相邻的节点，并且将大大增加其他数据包被阻塞的可能性，因此使死锁问题变得更加复杂，并且在网络负荷较重时，时延将变得不可预测。

2.2.2 地址空间

上面介绍了常见互联结构的分类，在弄清系统内各模块物理连接关系的同时，进一步了解目标系统的硬件架构，还需要从软件的角度看待各个组成模块，即对于地址映射表有所认识。

对于 SoC 来说，软件通常运行在处理器上以管理 SoC 片上的所有资源，如 DDR 存储控制器、中断控制器、定时器以及各种外设接口控制器。处理器需要对指定的系统模块进行操作，无论何种互联结构，都存在“地址”的概念，以区分互联结构上不同的系统模块。地址映射表即 SoC 片上所有资源的地址列表。需要注意的是，地址映射表由互联结构的设计决定，因此可能存在不同的地址列表，对

于交叉连接、片上网络上不同的处理器来说，同一系统模块的地址可能不同。我们在本节介绍的地址空间将主要针对传统的总线架构，毕竟为了保证软件访问芯片内各个设备的一致性，大多数芯片依然采用了传统的编址方法。

对于 x86 体系架构处理器来说，存在着两种地址空间，分别是内存空间与 I/O 空间。

内存空间作为主要的地址空间，通常包括 DRAM 等存储器。在 32 位系统中，内存空间的大小为 4GB。需要注意的是，在实际的系统中，可能并没有配置 4GB 的存储器，因此，地址空间并没有被完全映射，存在保留的地址空间片段（也就是没有实际的存储器对应这些地址空间）。对于内存空间的访问通常通过 MOV 等内存读写指令进行。

与内存空间相比，I/O 空间要小得多，只有 64KB。I/O 空间只能通过 IN/OUT 指令进行访问。通过 I/O 空间来访问 I/O 设备的延时较大，通常情况下，程序员应避免使用 I/O 空间。通过 I/O 空间来访问 I/O 设备属于早期设计，在现代设计中，I/O 设备通常被映射到内存空间上，通过内存空间来访问 I/O 设备。

嵌入式系统中常见的 ARM、PowerPC 等体系结构都采用这种统一的地址空间映射。也就是说，I/O 设备的相应配置寄存器、内部缓冲区或 FIFO 都统一映射到内存空间中的一段地址空间。

当处理器产生读写操作时，目标地址经由系统的地址译码器（通常在互联结构上）进行译码，互联结构根据译码结果将读写命令转发到对应的设备上，设备响应命令或返回数据，完成传输。如上所述，在现代设计中，内存空间包括存储器及 I/O 设备，处理器可以通过统一的访存指令（对于 ARM 架构而言就是 Load/Store 指令）经由 DRAM 控制器直接访问 DRAM 存储器对应的地址空间，也可以通过读写以太网控制器的硬件寄存器来收发数据包。

在嵌入式系统中，SoC 的地址映射表通常是固定的，所有模块在 SoC 设计时就分配有相应的地址空间，系统软件设计者，特别是驱动开发人员首先需要了解所要操作模块的地址。由于不同的 SoC 可能拥有不同的地址映射表，固定的地址映射表使得开发一套适用于不同 SoC 的驱动软件较为困难。通常情况下，嵌入式系统软件需要有针对性地设计及优化。我们将在本节的后续部分介绍 SEP4020 芯片的地址映射，感兴趣的读者还可以参阅第 5 章 5.5.3 节中关于 OMPA4460 芯片的地址映射。

1. SEP4020 地址映射表

地址映射表是芯片用户手册的重要内容，下面看看 SEP4020 的主要外设以及外部存储器的地址映射表，如表 2-1 所示。

地址段 0x04000000~0x07FFFFFF 为片上存储器 ESRAM（Embedded SRAM，学术界有时将这种集成在片上的静态存储器称为 Scratch Pad Memory，SPM）所占用的地址空间，这是 SEP4020 处理器在硅片上集成的一块 64KB SRAM。CPU 访问片上 ESRAM 是不需要通过 EMI 控制器的，CPU 可以直接通过总线访问 ESRAM 存储器。由于集成在片上，对这块存储器的读写速度基本上可以做到零等待，系统软件工程师可以将对访存延迟敏感的数据或代码存放在这里以优化系统的性能。基于 SPM 的优化问题，第 5 章将有比较详细的介绍。

地址段 0x10000000~0x1FFFFFFF 是 SEP4020 芯片的一些功能模块的寄存器所占用的空间，如中断控制器（INTC）、PMC、实时时钟与看门狗（RTC/WD）、定时器（TIMER）、脉宽调制器（PWM）、串行口（UART0/1/2/3）、同步串口（SSI）、音频接口（I2S）、MMC/SD 卡控制器、IC 卡控制器（SMC0/1）、USB、通用 IO（GPIO）、外部存储器接口（EMI）、DMA 控制器、LCD 控制器、以太网控制器（MAC）等模块的寄存器。底层软件人员通过 Load/Store 指令（当然也可以通过 C 语言的指针操作）读写这些模块的寄存器来实现对于它们的控制和数据传输。

表 2-1 SEP4020 缺省状态下的地址映射表

地址段	描述	地址段容量
0x00000000~0x03FFFFFF	EMI（nCSA）	64MB
0x04000000~0x07FFFFFF	ESRAM	64KB
0x08000000~0x0FFFFFFF	保留	
0x10000000~0x1000FFF	INTC	4KB
0x10001000~0x10001FFF	PMC	4KB
0x10002000~0x10002FFF	RTC/WD	4KB
0x10003000~0x10003FFF	TIMER	4KB
0x10004000~0x10004FFF	PWM	4KB
0x10005000~0x10005FFF	UART0	4KB
0x10006000~0x10006FFF	UART1	4KB
0x10007000~0x10007FFF	UART2	4KB
0x10008000~0x10008FFF	UART3	4KB
0x10009000~0x10009FFF	SSI	4KB
0x1000A000~0x1000AFFF	I2S	4KB
0x1000B000~0x1000BFFF	MMC/SD	4KB
0x1000C000~0x1000CFFF	SMC0	4KB
0x1000D000~0x1000DFFF	SMC1	4KB
0x1000E000~0x1000EFFF	USB	4KB
0x1000F000~0x1000FFFF	GPIO	4KB
0x10010000~0x10FFFFFF	保留	
0x11000000~0x11000FFF	EMI	4KB
0x11001000~0x11001FFF	DMAC	4KB
0x11002000~0x11002FFF	LCDC	4KB
0x11003000~0x11003FFF	MAC	4KB
0x11004000~0x11004FFF	保留	
0x11005000~0x11005FFF	AMBA	4KB
0x11006000~0x1FFFFFFF	RESERVED	
0x20000000~0x23FFFFFF	EMI（nCSA）	64MB
0x24000000~0x27FFFFFF	EMI（nCSB）	64MB
0x28000000~0x2BFFFFFF	EMI（nCSC）	64MB
0x2C000000~0x2FFFFFFF	EMI（nCSD）	64MB
0x30000000~0x33FFFFFF	EMI（nCSE）	64MB
0x34000000~0x37FFFFFF	EMI（nCSF）	64MB
0x38000000~0x3FFFFFFF	保留	

0x20000000~0x37FFFFFF 这段地址空间是分配给外部存储器的地址空间，当 CPU 或其他总线主设备访问该地址空间时，总线译码器将首先选通外部存储器接口（EMI）模块，而 EMI 模块将根据这个访问地址进一步译码并选通用于连接片外存储器的某个片选信号。SEP4020 的 EMI(External Memory Interface) 模块共有 6 个可以配置地址空间的片选信号输出：nCSA，nCSB，nCSC，nCSD，nCSE，nCSF。每个片选空间最大支持 64MB。所有片选都可以配置为 8 位或 16 位宽的 SRAM 时序（CSA 只能支持 16 位 SRAM，并且当任一片选被配置为 SRAM 时序时，其能访问的最大有效地址为 16MB），

CSE 和 CSF 可以配置为支持 16 位宽的 SDRAM 时序（每个片选支持的最大有效地址为 64MB）。可以通过配置各个片选的基址寄存器（这些寄存器的地址位于 EMI 模块所占用的地址空间 0x11000000~0x11000FFF 内）设置各个片选信号的读写属性、数据宽度以及基址空间。如果用户不设置片选所对应的基址，EMI 的 6 个片选信号所对应的默认地址范围为 0x20000000~0x37FFFFFF。

需要注意的是，在某些情况下，地址映射表是可以动态改变的，这就是所谓的“地址重映射（Remap）”机制。在 SEP4020 的地址空间中，地址空间 0x00000000~0x03FFFFFF 可以映射到不同的存储器模块上，包括片外 NorFLASH、片外 SRAM 或者 SDRAM 存储器、ESRAM 等。系统设计人员可以通过设置 SEP4020 的 3 个引脚（SYS_SETUP[2:0]）的高低电平组合，选择系统是从 Nor Flash 启动还是从 Nand Flash 启动。如果 SYS_SETUP[2:0] 这 3 根引脚全部拉低为低电平 0，系统将从 Nor Flash 启动（注意该 Nor Flash 芯片必须连接在 CSA 片选），此时 0x00000000~0x03FFFFFF 地址段将被映射到 CSA 片选的地址空间。而原来 CSA 的缺省地址段 0x20000000~0x23FFFFFF 也同时有效，也就是 CSA 所映射的 Nor Flash 同时具备两个不同的地址段。将零地址映射到 Nor Flash 是因为 ARM 处理器复位后从零地址开始取第一条指令，因为中断向量表存放在这个地址。如果系统被设置为从 Nand Flash 启动，零地址的映射会更加复杂，感兴趣的读者可以参阅 SEP4020 处理器的用户手册。除了将零地址段映射到 CSA 外，系统程序员还可以通过配置 EMI 中重映射配置寄存器（REMAPCONF）将零地址段映射到其他任一片选。基于这个特性，程序员可以在系统启动完成后，将零地址段映射到 CSE 或 CSF 片选。因为这两个片选可以被配置为 SDRAM，这样程序员就可以在 SDRAM 空间内构件中断向量表了，这对于像 Linux 这样的操作系统而言，无疑大大增加了实现的灵活性。

2.2.3 常见互联结构接口协议

规划一个 SoC 设计，首先要考虑如何把各种功能模块集成起来。SoC 的复杂度不断提升，使得 IP 核复用技术在 SoC 设计中变得尤为重要。

IP 核将一些在数字电路中常用、但比较复杂的功能块（如 SDRAM 控制器、PCI 接口等）设计成可修改参数的模块。IP 核复用是设计人员赢得上市时间的主要策略。随着 SoC 的规模越来越大，设计越来越复杂（IC 的复杂度以每年 55% 的速率递增，而设计能力每年仅提高 21%）。复用 IP 核能避免重复劳动，大大减轻工程师的负担，因此使用 IP 核是一个发展趋势。

构建 SoC 系统即设计片内互联结构的过程，具体来说也就是解决各功能模块间的相互通信问题，包括时序和协议等方面。IP 核是为了易于重用而专门设计的，设计的理想目标是即插即用，采用业界通用的接口标准可以使 IP 核具有较好的可移植性。

下面将重点介绍 ARM 公司的 AMBA（Advanced Microcontroller Bus Architecture）总线规范，同时简单概述 Silicore 公司设计的 Wishbone 规范和 Altera 公司的 Avalon 规范。

AMBA 规范是 ARM 公司设计的一种用于高性能嵌入式系统的互联结构接口标准。它独立于处理器和制造工艺技术，增强了各种应用中外设和系统宏单元的可重用性。AMBA 规范是一个开放标准，可免费从 ARM 公司获得。目前，AMBA 规范得到众多第三方支持，被 90% 以上的 ARM 合作伙伴采用，在基于 ARM 处理器内核的 SoC 设计中，已经成为广泛支持的现有互联标准之一。

1. AMBA 2 接口协议

AMBA 2 规范于 1999 年发布，规范主要包括 AHB（Advanced High performance Bus，高级高性能总线）和 APB（Advanced Peripheral Bus，高级外设总线）。AMBA 2 规范不仅包括相应的接口协议，还描述了接口模块的互联体系，对芯片上模块之间的互联具有重要意义。

基于 AMBA 的微控制器通常包括一个高性能系统总线 AHB，用于高性能 ARM 处理器（CPU 内核）、DMA 与高带宽片上 RAM、高带宽存储器接口之间的数据传输，同时通过一个桥接器连接到较窄的、挂接较低带宽外设的 APB 总线。图 2-13 表示包含 AHB 和 APB 的一个典型系统。

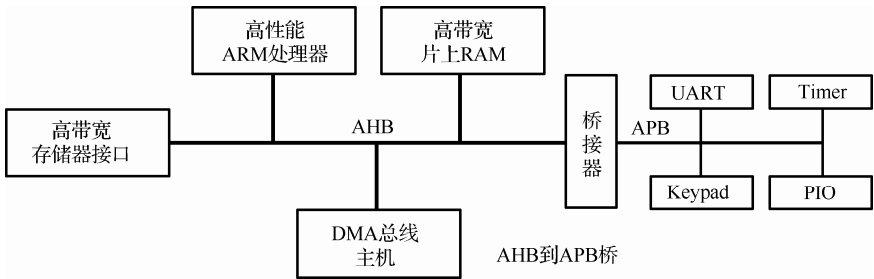


图 2-13 包含 AHB 和 APB 的一个典型系统

AHB 的主要特点如下：

- 高性能、高带宽，采用两级流水线结构，单时钟上升沿操作；
- 支持复杂的总线拓扑结构；
- 支持多主机，可配置 32 位~128 位总线数据位宽；
- 无三态应用，读写数据线相互独立；
- 支持猝发（Burst）传输；
- 支持分段传输；
- 支持字节、半字和字的传输。

AHB 主要用于高性能模块（如 CPU、DMA 和 DSP 等）之间的连接，基于 AHB 总线结构的系统由主设备（Master）、从设备（Slave）、基础结构（Infrastructure）三部分组成，如图 2-14 所示。AHB 总线上的传输都由主设备发出，从设备负责回应。基础结构由仲裁器、主设备到从设备的多路选择器、从设备到主设备的多路选择器、译码器所组成。

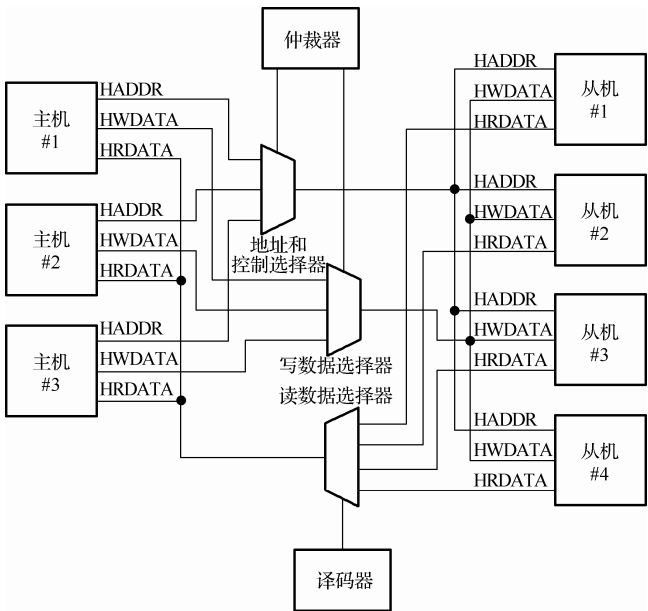


图 2-14 AHB 总线协议

- ① AHB 主设备：总线主设备能提供地址和控制信息来对数据行读写操作，即主设备发起数据传输。
- ② AHB 从设备：总线从设备对主设备发起的数据传输做出响应，响应读写数据操作并返回状态信号（成功、失败或者等待）给主设备来完成数据传输。
- ③ AHB 仲裁器：总线仲裁器负责选择合适的主设备拥有对总线的控制权，即被选中的主设备可以发起数据传输。总线仲裁器采用合适的算法（高优先级或公平访问等）来满足不同系统需要。
- ④ AHB 译码器：AHB 译码器用于对每一次数据传输地址进行译码，同时在数据传输中向从设备给出一个选择信号。

主设备发出地址和控制信号来指示它们想执行的传输，仲裁器决定哪一个主设备能够获得对总线的控制权，译码器将根据主设备发出的地址，产生相应的从设备选择信号，以通知此次传输的从设备，译码器还将控制相应的选择器控制数据和控制信号的流向。

AHB 总线操作的典型过程如下：

- 在一个 AHB 总线数据传输之前，主设备必须已经从总线仲裁器那里获得总线控制权。
- 一个获得授权的主设备通过驱动地址和控制信号开始一次单次传输或者猝发传输。
- 一次传输由一个或多个地址相位和相应的数据相位组成。

AHB 信号列表如表 2-2 所示。

表 2-2 AHB 信号列表

名称	来源	描述
HCLK 总线时钟	时钟源	为所有总线传输提供时钟，所有信号时序都和 HCLK 的上升沿相关
HRESETn 复位	复位控制器	总线复位信号低有效并用来复位系统和总线。这是唯一的低有效的信号
HADDR[31:0] 地址总线	主设备	32 位系统地址总线
HTRANS[1:0] 传输类型	主设备	表示当前传输的类型，可以是不连续、连续、空闲和忙
HWRITE 传输方向	主设备	当该信号为高时表示一个写传输，为低时表示一个读传输
HSIZE[2:0] 传输大小	主设备	表示传输的大小，典型情况是字节（8 位）、半字（16 位）或者是字（32 位）。协议允许最大的传输大小可以达到 1024 位
HBURST[2:0] 猝发类型	主设备	表示传输是否组成了猝发的一部分。支持 4 个、8 个或者 16 个节拍的猝发传输并且猝发传输可以是增量或者是回环
HPROT[3:0] 保护控制	主设备	提供总线访问的附加信息并且主要是给那些希望执行某种保护级别的模块使用的。这个信号指示当前传输是否为预取指令或者数据传输，同时也表示传输是保护模式访问还是用户模式访问。对带存储器管理单元的总线主机而言这些信号也用来指示当前传输是高速缓存的（cache）还是缓冲的（buffer）
HWDATA[31:0] 写数据总线	主设备	写数据总线用来在写操作期间从主设备到总线从设备传输数据
HSELx 从设备选择	译码器	每个 AHB 从设备都有自己独立的从设备选择信号并且用该信号来表示当前传输是否是打算送给选中的从设备。该信号是地址总线的简单组合译码
HRDATA[31:0] 读数据总线	从设备	读数据总线用来在读操作期间从总线从设备向总线主设备传输数据
HREADY 传输完成	从设备	当 HREADY 为高时表示总线上的传输已经完成，HREADY 为低时表示从设备需要插入一个等待周期
HRESP[1:0] 传输响应	从设备	传输响应给传输状态提供了附加信息。提供 4 种不同的响应：OKEY、ERROR、RETRY 和 SPLIT

图 2-15 表示一次简单的没有等待状态的数据传输。在数据传输过程中：

- 主设备在 HCLK 的上升沿之后将地址和控制信号驱动到总线上；
- 从设备在下一个 HCLK 时钟上升沿采样地址和控制信息；
- 在从设备采样地址和控制信息后，它开始驱动相应的响应，主设备将在第三个 HCLK 时钟上升沿采样从设备的响应信息。

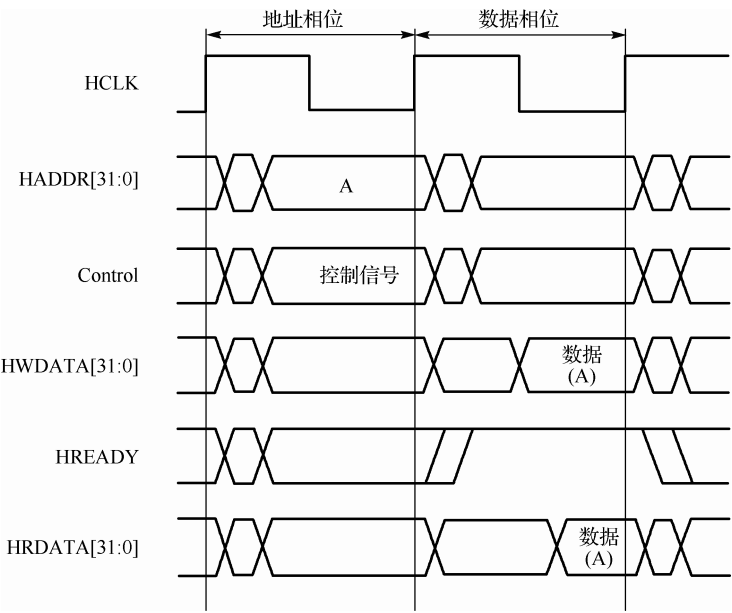


图 2-15 一次简单的没有等待状态的数据传输

以上的例子显示了一次数据传输中不同的时钟周期时地址和数据的状态。实际上，任何一次数据传输中，此次的地址状态和上一次传输的数据状态是同时存在的。这种地址和数据的交叠是总线传输中最基本的流水线技术。如图 2-16 所示。在该图中，主设备在第一个时钟周期发出了地址 A 和与之对应的控制 A；第二个时钟周期从设备响应了本次请求，并完成了数据的传输（图中的数据 A），第二个时钟周期主设备紧接着还发出了新的地址 B 和控制 B；第三个周期，从设备未准备好，将 HREADY 信号保持为低，主设备延缓了对数据总线的锁存，但依然继续发出了新的地址请求 C 和控制信号 C；第四个周期，从设备拉高 HREADY，主设备锁存数据 B。因此在 AMBA 的传输中地址和数据在时间上是交叠（overlap）的，这提高了总线的吞吐量。

AHB 协议支持猝发（Burst）传输，猝发传输即连续进行多个连续地址的数据传输。

猝发传输有什么作用？通常来说，主设备发出的传输命令到达从设备，由于仲裁等原因，传输会产生一定的延时。如果主设备读一个数据，处理一个数据的话，由于延时的存在，这样的处理速度无疑是极低的。考虑到指令、数据具有一定的空间连续性，诸如 CPU 之类的主设备通常设计有高速缓存（指令 Cache 和数据 Cache），通过猝发传输连续读/写数据，可以减小延时带来的影响，以提高性能。

另一方面，从设备接收命令到返回数据也可能存在一定的延时，特别是片外存储器。片外存储器通常需要搭配存储控制器共同工作，作为总线从设备的控制器接收传输命令，再转化为相应的控制信号读/写相连的存储器，信号转化及存储器返回数据的过程中不可避免将产生延时。对于一些支持猝发传输的存储器来说，比如 DRAM 存储器，控制器发出一次猝发传输后，经过一定的延时后，DRAM 可以连续地返回数据，除了第一个数据的传输需要若干周期的延时外，其后的数据可以连续获得。

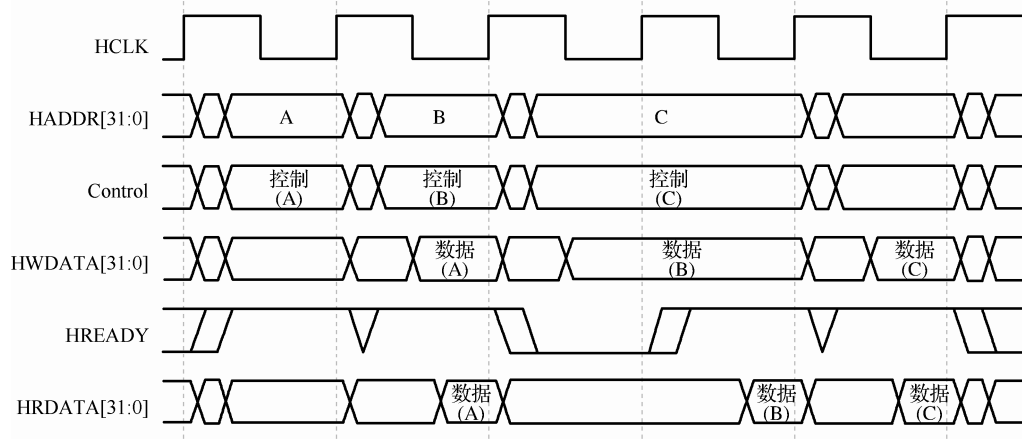


图 2-16 AHB 的流水传输

AHB 协议定义了单次传输，4 拍、8 拍和 16 拍的猝发传输，以及未定长度的猝发传输。AHB 协议支持增量（Incrementing）猝发传输和回环（Wrapping）猝发传输两种猝发传输方式：

- ① 增量猝发访问连续地址并且猝发中的每次传输地址仅是前一次地址的一个增量；
- ② 对于回环猝发，如果传输的起始地址并未和猝发传输的总字节数对齐，那么猝发传输地址将在到达边界处回环。例如，一个 4 拍回环猝发的字（4 字节）访问将在 16 字节边界回环。因此，如果传输的起始地址是 0x38，那么它将包含 4 个地址：0x38、0x3C、0x30 和 0x34。

一次 4 拍回环猝发传输如图 2-17 所示，图中 HTRANS[1:0] 信号是传输状态信号，有 4 种状态：非连续/NSEQ（表示一次单个数据的传输或者猝发传输的第一个数据）、连续/SEQ（表示猝发传输接下来的数据）、空闲/IDLE（表示没有进行数据传输）、忙/BUSY（表示在猝发传输的过程中主设备忙，需要推迟一个周期传输数据）。图 2-17 中 HBURST[2:0] 信号是猝发传输类型信号，有 8 种猝发传输类型：单一传输/SINGLE、未指定长度的增量猝发/INCR、WRAP4/4 拍回环猝发、INCR4/4 拍增量猝发、WRAP8/8 拍回环猝发、INCR8/8 拍增量猝发、WRAP16/16 拍回环猝发、INCR16/16 拍增量猝发。

下面简单介绍一下 APB 总线协议。

高级外设总线（APB）是 AMBA 总线结构的一部分，为降低功耗和接口复杂性做了优化。APB 可以用来连接低带宽的外设。APB 的主要特点如下：

- 非流水线操作；
- 接口简单；
- 低功耗；
- 适合外围低速设备。

图 2-18 表示 APB 基本的写传输。地址（PADDR）、写信号（PWRITE）、选择信号（PSEL）、写数据（PWDATA）在时钟上升沿之后同时有效，表明一次写传输开始。传输的第一个时钟周期称为 SETUP 周期。在下一个时钟上升沿使能信号（PENABLE）有效，紧接着的就是 ENABLE 周期。地址、写数据和控制信号在此期间保持有效。传输在 ENABLE 周期结束时完成，此时使能信号失效。选择信号也将变成低电平，除非当前传输之后紧接着另一个传输。

图 2-19 表示 APB 基本的读传输。地址、写信号、选择信号的时序和写传输一样。在读传输的情况下，从设备必须在 EENABLE 周期提供数据。读数据（PRDATA）在 ENABLE 周期结束的时钟上升沿被主设备采样。

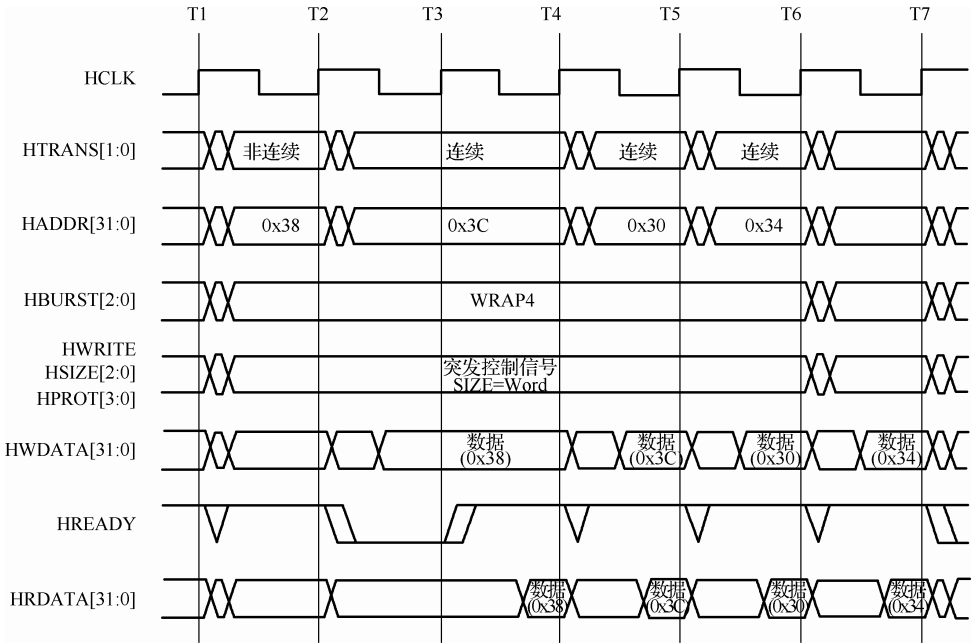


图 2-17 一次 4 拍回环猝发传输

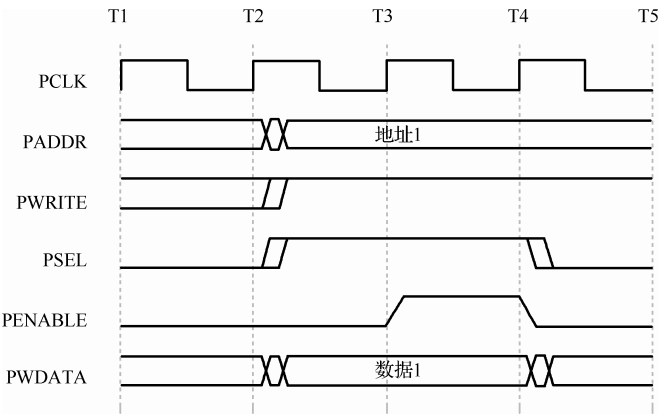


图 2-18 APB 写操作

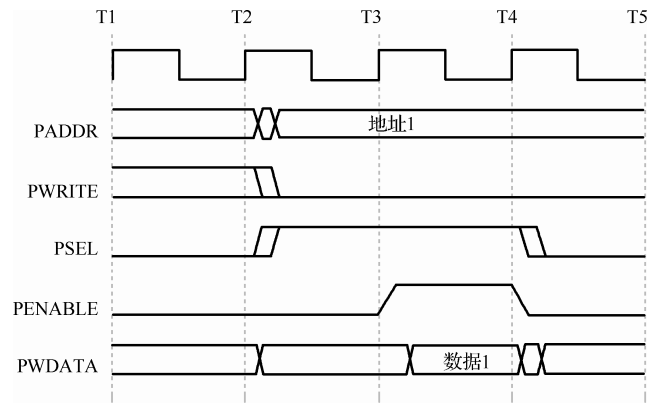


图 2-19 APB 读操作

2*. AMBA 3 接口协议

对于简单的接口传输, AHB 协议简洁、高效, 但是对于超高速接口传输来说, 性能仍有不足。特别是对于 DRAM 来说, AHB 固有的一个时钟周期的流水线特点不能有效发挥 DRAM 的数据传输性能。

数据在 DRAM 中是以矩阵形式存储的, 通过行、列地址对数据进行定位传输。对于 DRAM 的读/写操作, 就是一次对于存储单元的寻址。相关的列地址被选中之后, 将会触发数据传输, 但从存储单元中输出到真正出现在内存芯片的 I/O 接口之间还需要一定的时间(数据触发本身就有延迟, 而且还需要进行信号放大), 这段时间就是非常著名的 CL (CAS Latency, 列地址脉冲选通延迟)。如果要连续读/写就要对当前存储单元的下一个单元进行寻址, 也就是要不断地发送列地址与读/写命令。由于读/写延迟相同, DRAM 的设计机制可以让数据的传输在 I/O 端是连续的, 如图 2-20 所示。

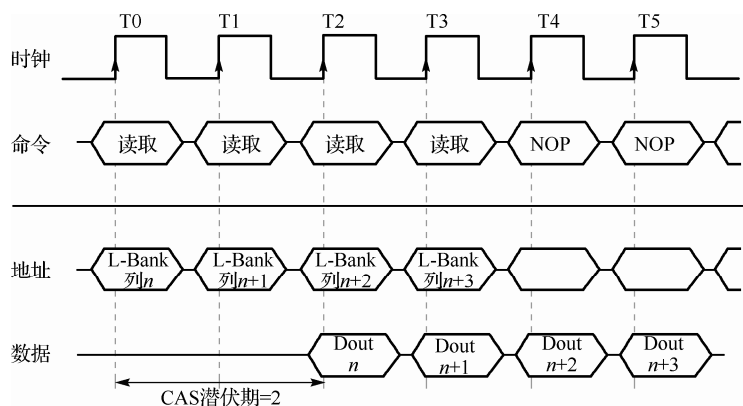


图 2-20 DRAM 在 I/O 端的数据传输机制

由于 AHB 协议只能实现一个时钟周期的流水操作, 主机在第一个数据未返回的情况下发出第二个读取命令, 直到第一个数据返回, 才能发出第三个读取命令, 这将使得 DRAM 无法实现连续操作, 读/写效率较低。

为了解决这个问题, AMBA3 标准支持 Outstanding 传输, 也就是指主机发起一次传输后, 可以不用等待传输完成就发起下一次传输。Outstanding 传输可以实现多个传输的并行化, 有效提高系统性能。注意, Outstanding 传输(有时也被译为乱序传输)和 AMBA2 中支持的猝发传输不是一个概念, 在 AMBA2 中, 一次单独的传输或是一个猝发传输也被称为一个总线交易(Transaction), 在一个交易完成前, 总线主设备是不能发起另一个交易的。但是 Outstanding 传输允许交易之间可以交叠(Overlap), 也就是一个交易还没完成就可以发起另一次新的交易。

AMBA 3 规范定义了 4 个接口协议, 包括:

- AMBA 3 AXI (Advanced eXtensible Interface): 针对要求高数据吞吐量、高带宽通信的设备。
- AMBA 3 AHB: 针对性能要求较低, 不需要 AXI 接口的简单设备。
- AMBA 3 APB: 针对要求低门数、低功耗的设备。
- AMBA 3 ATB (Advanced Trace Bus): 针对执行片上测试和调试访问的数据集中处理组件。

作为 AMBA 3 规范中一个新的高性能协议, AXI 技术丰富了 AMBA 2 标准内容, 满足超高性能和复杂的片上系统的设计需求。AXI 协议的特点如下:

① 独立的读地址/控制通道、写地址/控制通道、读数据通道、写数据通道、写响应通道, 每个通道单独控制, 数据只能单方向传输, 不再复用控制信号。虽然信号多了, 但是易于实现高效、高速、高性能的接口传输, 如图 2-21 所示。

- ② 支持 Outstanding 传输，可以实现深度不固定的流水传输。
- ③ 猝发传输只需要发出首地址和传输参数，其余地址由从设备根据传输参数计算得到，这可以减少总线上的能量消耗。
- ④ 支持乱序传输，能够提高 DRAM 的传输效率。
- ⑤ RTL 实现时可以通过增加“寄存器片”，更易于时序收敛，提高逻辑综合的效率。

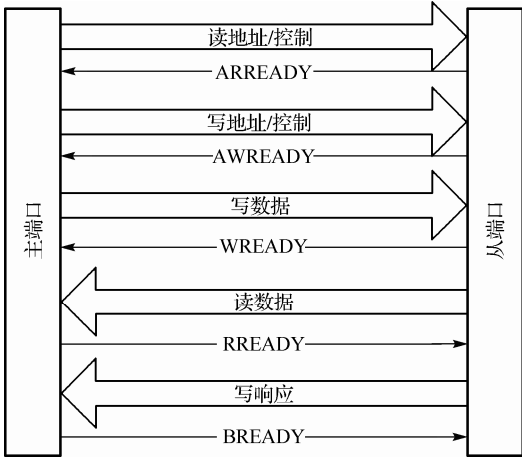


图 2-21 AXI 协议

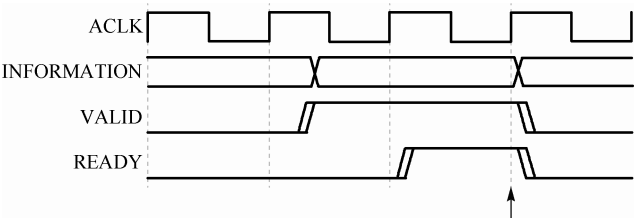


图 2-22 基于 AXI 协议的典型传输过程

基于 AXI 协议的典型传输过程如图 2-22 所示。图中：

- 每个通道中的源端提供 VALID 信号，表明命令或者数据准备好。
- 每个通道中的目的端提供 READY 信号，表明可以接收命令或者数据。
- 通道的 VALID 和 READY 同时有效，表明一次命令或者数据的交易有效。
- 读写数据通道提供一个 LAST 信号，表明是传输的最后一拍数据。
- 写响应通道用来通知源端数据真正被写入目的端。

需要注意，源端产生的 VALID 信号一定不能依赖于相应的 READY 信号。因为目的端可以等待 VALID 信号有效后再有效 READY 信号，这种情况下，VALID 信号依赖于 READY 信号将产生死锁问题，导致传输中断。

从设备可以等待 VALID 信号有效后再有效 READY 信号，也可在 VALID 信号有效之前就默认有效 READY 信号，以提高传输效率，如图 2-23 所示。

AXI 协议依靠通道间的握手来实现数据传输，规定的 5 个通道之间存在如下两条必需的依赖关系：

- ① 读数据必须依赖于（即落后于）相应的读地址/控制信号。如图 2-24 所示，单箭头指向的信号可以在箭头源信号有效之前或之后有效，没有固定的依赖关系；双箭头指向的信号必须在箭头源信号

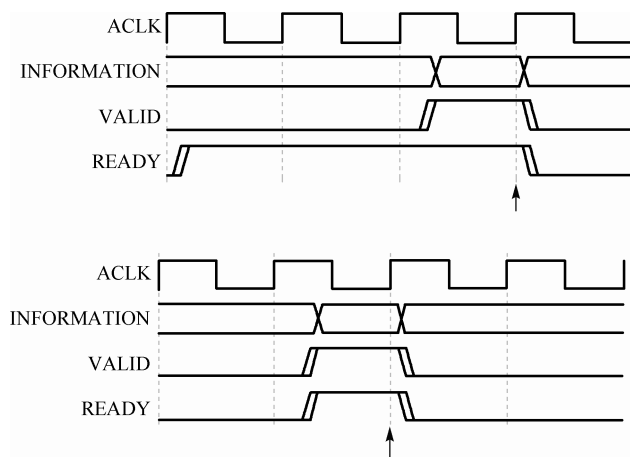


图 2-23 VALID 信号和 READY 信号

有效之后有效。图中 ARVALID 和 ARREADY 分别指读地址通道的 Valid 信号与 Ready 信号，而 RVALID 和 RREADY 则指读通道的 Valid 信号与 Ready 信号。

- 从设备可以等待 ARVALID 信号有效之后，再有效 ARREADY 信号；
- 从设备必须等待 ARVALID 信号与 ARREADY 信号均有效之后，才能通过有效 RVALID 信号返回数据。

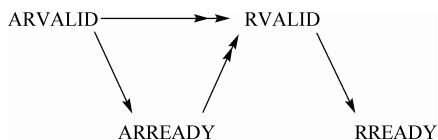


图 2-24 读数据依赖于相应的读地址/控制信号

② 写响应信号必须依赖于（即落后于）相应的写数据信号。如图 2-25 所示，单箭头指向的信号可以在箭头源信号有效之前或之后有效，没有固定的依赖关系；双箭头指向的信号必须在箭头源信号有效之后有效。

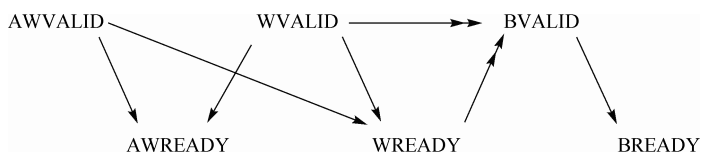


图 2-25 写响应信号依赖于响应的写数据信号

- 主设备根据需要有效 AWVALID 信号与 WVALID 信号，不能等待 AWREADY 信号与 WREADY 信号；
- 从设备可以等待 AWVALID 信号或是 WVALID 信号有效之后，再有效 AWREADY 信号；
- 从设备可以等待 AWVALID 信号或是 WVALID 信号有效之后，再有效 WREADY 信号；
- 从设备必须等待 WVALID 信号或是 WREADY 信号有效之后，再有效 BVALID 信号。

3*. AMBA 4 接口协议

AMBA 4 规范在 AMBA 3 规范的基础上另外新增了 3 个接口协议，包括 AXI4、AXI4-Lite 以及 AXI4-Stream，其中 AXI4 协议是对 AXI3 的更新，可提高互联的性能和利用率。它包括以下增强功能：

- ① 对于增量猝发传输，猝发长度提高到 256 次传输。
- ② 支持服务质量（QoS）。
- ③ 支持多区域接口。

AMBA4 规范的细节在此不再详述，有兴趣的读者可以阅读最新的 *AMBA AXI and ACE Protocol Specification*，进行更深入的了解。

4. 其他接口协议*

Wishbone 规范最先是由 Silicore 公司提出的，现在已被移交给 OpenCores 组织维护。由于其开放性，现在已有不少的用户群体，特别是一些免费的 IP 核，大多数都采用 Wishbone 标准。灵活性是 Wishbone 规范的一个优点。用户可以按需要自定义 Wishbone 标准，如字节对齐方式和标志位（TAG）的含义等，还可以加上一些其他的特性。

Avalon 规范是 Altera 公司设计的用于 SOPC（System On Programmable Chip，可编程片上系统）中，连接片上处理器和其他 IP 模块的一种简单的接口协议，规定了主部件和从部件之间进行连接的端口和通信的时序。

Wishbone 规范的简单性和灵活性受到广大 SoC 设计者的青睐，因为它是完全免费的，并有丰富的免费 IP 核资源。Avalon 规范主要用于 Altera 公司系列 PLD 中，最大的优点在于其配置的简单性，可由 EDA 工具快速生成，受 PLD 厂商巨头 Altera 极力推荐，其影响范围也不可忽视。

2.3 中央处理器

SoC 的核心是中央处理器（以下简称 CPU），CPU 执行应用程序，控制整个系统的正常运行。CPU 通常运行操作系统，通过操作系统来管理各个应用程序的执行。

在现代嵌入式系统 SoC 中，除 CPU 之外，通常还包括其他一些处理单元，这些处理单元针对指定功能专门设计，用以支持特定应用。像是在现代智能手机中，主处理器为 CPU，即应用处理器（Application Processor），运行面向用户的软件，诸如用户接口，以及各种各样的应用程序（网络浏览器、愤怒的小鸟、导航服务，等等）；同时，基带处理器（Baseband Processor，通常是一个特定的 DSP 处理器）用以处理无线网络协议栈。在其他一些嵌入式系统 SoC 中，还集成有用于音频及视频处理的专用处理器。运行在这些专用处理器上的软件通常称为固件（fireware），一般无需运行操作系统，当然有时这些固件也包含了一个专用的小操作系统。

处理器需要其他硬件的配合工作才能实现多种应用。对于运行多任务操作系统的处理器来说，必需的硬件支持包括：

- 存储子系统：用以存储指令及数据读/写。
- 中断控制器：管理 SoC 中其他设备或外部硬件送给处理器的中断信号。
- 定时器：多任务操作系统一般依赖定时器产生相应的中断信号，用以触发操作系统的调度执行。
- I/O 设备：诸如显示控制器、网络接口控制器、输入（触摸、键鼠）接口控制器等。

CPU 处于 SoC 的核心地位，通过互联结构与其他设备进行交互，实现系统功能。关于 SoC 中 CPU 内核的详细介绍见第 4 章。

2.4 中断控制器

CPU 通过与一系列的输入/输出设备进行交互，实现功能。这些外设 CPU 的控制下与物理世界产生交互，系统中的外设数量通常远多于 CPU 数量，但需要在一段时间内控制多个外设，处理外设需求。因此外设需要一定的机制来得到 CPU 的服务。一种方法是让 CPU 不停地轮询系统所有外设（其实是运行在 CPU 上的软件进行查询），查询是否有外设需要操作。这种方法将占用 CPU 的计算资源，

影响系统的整体性能。另一种方法可以避免 CPU 对于外设的轮询操作，而是采用中断的方式，异步地处理外设请求。需要 CPU 操作的外设发起中断信号，提示处理器中断当前操作，服务需要操作的外设。当多个中断信号有效时，CPU 需要依靠中断控制器决定需要优先服务的中断信号，对相应的外设进行操作。

中断控制器汇集 SoC 系统中的所有的硬件中断信号^①，根据一定的配置管理这些中断信号，最终产生有效的中断信号，提示 CPU 进行中断处理。中断控制器最基本的功能是将中断信号路由到 CPU，当 CPU 响应这个中断请求时，首先将通过软件读取中断控制器中的中断状态寄存器，获得产生中断的中断源，从而运行相应的中断服务函数，执行相应的操作。

图 2-26 是中断控制器的基本实现形式，包括 3 个处理器可读/写的寄存器，这些寄存器的每一位分别对应一个中断源。

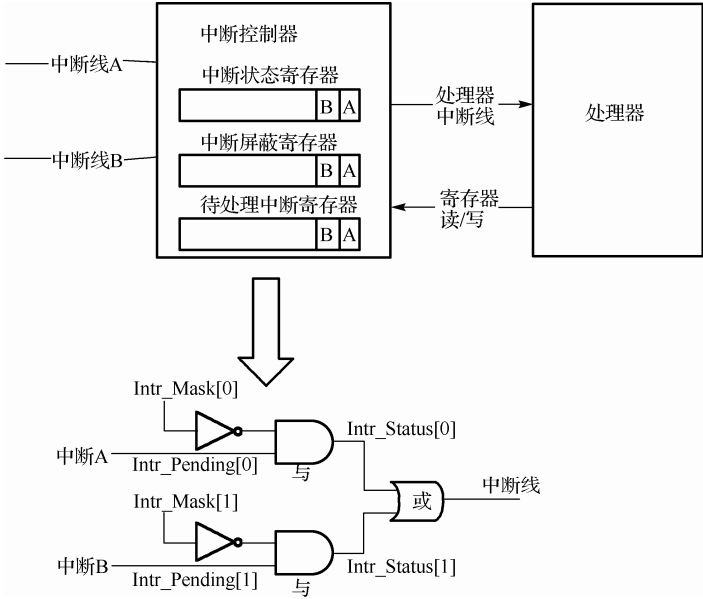


图 2-26 中断控制器

- ① 待处理中断寄存器（Intr Pending）：待处理中断寄存器反映当前所有中断源的中断状态。中断信号有效时，相应的待处理中断位将置位；中断源没有待处理的中断时，相应的待处理中断位将清零。
- ② 中断屏蔽寄存器（Intr Mask）：中断屏蔽寄存器可以屏蔽中断源产生的中断信号。某一中断屏蔽位有效后，中断源产生的中断信号将无法到达处理器，使得处理器不会被相应中断源的中断信号所影响。
- ③ 中断状态寄存器（Intr Status）：中断原始状态寄存器反映的是屏蔽前的中断源状态。中断状态寄存器反映的是屏蔽操作后的中断信号。

屏蔽操作后，如果存在有效的中断信号，中断控制器将通过处理器的中断信号线通知处理器产生中断操作。一旦中断信号线有效，处理器将保存当前处理器部分状态，然后读取中断状态寄存器，根据预定的算法，决定优先处理的中断信号。处理器处理完中断信号后，将控制中断源清除中断信号，处理器退出中断处理过程，继续中断前的操作，或是响应下一个中断信号。

① 本书 4.2.5 节和 7.2.5 节比较详细地介绍了 ARM 处理器如何响应中断和软件通常是如何处理中断的。

如上所述，如果使用处理器读取中断状态寄存器，然后根据算法扫描中断状态寄存器的每一位，决定优先级最高的中断信号进行相应的中断操作，这一过程将增加待处理中断的等待时间。为了解决这一问题，可以设计专用的硬件模块，根据硬件固化的优先级策略处理有效的中断信号，决定优先级最高的中断信号，产生相应的中断源编号。这样，一旦中断信号线有效，处理器就可以直接读取中断源编号，快速运行相应的中断服务函数，减少中断响应时间。

中断信号可以来自 SoC 中集成在片上的设备（比如 Timer、UART、SPI 等），也可以来自外部端口，诸如可以产生中断信号的通用输入/输出端口（General-Purpose Input/Output, GPIO），GPIO 的介绍请见第 6 章。中断信号有两种触发形式，通常中断控制器同时支持多种触发方式，用户只需要在中断控制器中的相应寄存器中配置该信号的触发方式即可。

① 电平触发（高、低电平）：中断信号的逻辑值决定中断是否有效。高电平触发时，中断信号置位时表明中断请求有效，中断信号为低电平时表明没有中断请求。低电平触发时则相反。片上设备中多使用电平触发方式，多个设备的中断信号输出可以通过简单的逻辑操作，作用于最终的处理器中断信号线。

② 边沿触发（上升沿、下降沿或双沿）：中断信号的逻辑改变表明中断产生。上升沿触发时，中断信号由低电平变为高电平表明设备产生中断；下降沿触发时，中断信号由高电平变为低电平表明设备产生中断；双沿触发时，中断信号的电平变化都将产生中断。

嵌入式系统需要区分各个中断并执行不同的中断服务函数。电平触发中断将保持中断有效，等待处理器操作。中断服务函数需要执行特定的操作，控制外设清除（撤销）相应的中断信号，否则，外设将持续地中断处理器操作，导致正常的处理器程序无法运行。与电平触发中断相比，边沿触发中断可以实现中断信号的自动清除。外设产生中断的原因很多，使用边沿触发时，中断服务函数需要确保处理所有的中断信号。而使用电平触发则不需要注意这一问题，如果中断信号没有全部得到处理，中断信号线将持续有效，重复触发处理器中断（通常情况是当中断服务程序读中断控制器的中断状态寄存器时，中断控制器的硬件将自动撤销发给 CPU 的中断信号，而当中断服务程序读中断源设备的中断状态寄存器时，设备的硬件也将自动撤销发往中断控制器的中断信号。当然，有些硬件实现不会自动撤销信号，需要程序员显式地清除中断请求）。

2.5 存储子系统

存储子系统可能是嵌入式系统中除了 CPU 外最重要的系统了。应用的所有程序、数据、用户的数据都存放在存储子系统中。存储子系统是现代嵌入式系统的性能、成本和功耗的瓶颈。广义上来说，嵌入式系统的存储子系统包括从片上存储器到片外存储器构成的完整系统，其中片上存储器包括：CPU 内部的寄存器堆（Register File）、高速缓存（Cache 或 TCM^①，甚至是多级高速缓存）、片上草稿存储器（SPM, Scratch Pad Memory）以及相关缓冲存储器（Buffer）。片外存储器主要包括：主存储器（通常是 DDR SDRAM 或 SDRAM）和非易失存储器（通常是 Flash 存储器和 SD 卡）。这个由片上片外存储器所构成的复杂层次化系统称为存储架构（Memory Hierachy）。

存储子系统由存储器与相应的控制器组成。如图 2-27 所示，OMAP35x 系列 SoC 内部集成了 GPMC（General Purpose Memory Controller, 通用存储器控制器）、SDRC（SDRAM 存储器控制器）、MMC（MMC 接口兼容设备控制器）、可以连接 OneNAND 类型的 Flash 存储器、Low-power DDR 存储器、MMC/SD

① Tightly Coupled Memory，紧耦合存储器，一般指通过专用总线与 CPU 连接的高速片上存储器。

存储卡以及采用 CE-ATA 接口（兼容 MMC 接口）的 HDD（Hard Disk Drive，硬盘驱动器），共同组成嵌入式系统的存储子系统。

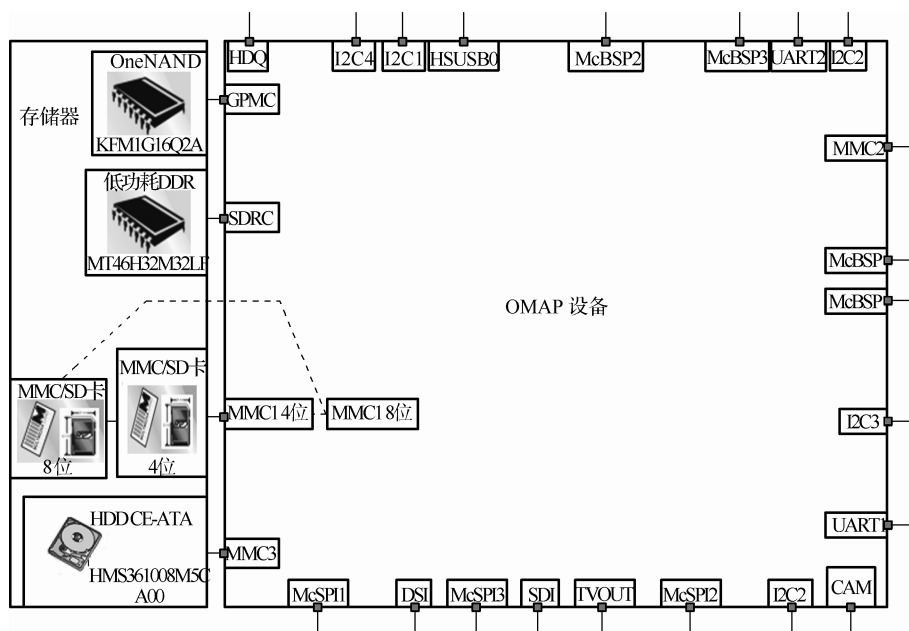


图 2-27 OMAP35x 系列 SoC 的存储子系统

存储器可分为易失性存储器（Volatile Memory）及非易失性存储器（Nonvolatile Storage）。

易失性存储器是指当电源断电后，所存储的数据便会消失的存储器。易失性存储器的主要类型为随机存取存储器（Random Access Memory，RAM），通常作为操作系统及应用程序运行时的临时存储介质，有时被称作内存、主存。RAM 可以分为 SRAM（Static Random Access Memory，静态随机存取存储器）和 DRAM（Dynamic Random Access Memory，动态随机存取存储器）两大类。DRAM 由于具有较低的单位容量价格，所以被大量采用作为系统主存，如图 2-27 所示的 Low-power DDR 存储器。

电源关闭时，RAM 不能保留数据，如果需要保存数据，就必须把它们写入到一个非易失性存储器中，例如计算机平台上的硬盘驱动器。除了保存用户的数据之外，非易失性存储器还用于保存系统运行的程序及相应数据。非易失性存储器如图 2-27 所示的 OneNAND 存储器、MMC/SD 卡以及硬盘驱动器。在嵌入式系统中（尤其是手持终端），受设备体积限制，硬盘并不常用，非易失性存储器多为 Flash 存储器（OneNAND 存储器属于 Flash 存储器）及 MMC/SD 存储卡。

关于嵌入式系统的存储子系统的详细介绍见第 5 章。

2.6 直接存储器访问（DMA）

常见的处理器与外设间的数据传输主要有 4 种方式：第一是 CPU 不断主动发起对外设的轮询，如果有数据则直接读取外设的相应 FIFO；第二是通过中断方式，当外设的 FIFO 中有数据时，将由外设发起中断（一般可以设为 FIFO 满中断或者 FIFO 半满中断），请求 CPU 在中断服务程序中将 FIFO 中的数据读取到内存缓冲区；第三种方法是通过直接存储器访问即 DMA（Direct Memory Access）进行数据传输，CPU 通过配置 DMA 控制器，由 DMA 代替 CPU 将外设 FIFO 中的数据传输到内存缓冲区或另外一个外设的输入 FIFO；第四种方法则是在需要数据传输的设备间建立专用的硬件通道，相互连

接的两个设备通过专用的硬件互联通道进行数据传输（这种情况类似于互联架构中的 P2P 连接，由于非标准且成本较高，一般使用较少）。本节将重点介绍 DMA 数据传输。

某个 I/O 设备需要数据传送时，首先由 CPU 配置相应的 DMA 控制器，作为总线主设备的 DMA 控制器（DMAC）发出总线请求信号，总线仲裁器将总线控制权交给 DMAC 后，DMAC 接管总线，并发出对该 I/O 设备数据端口的读控制信号和相应的地址，将设备的数据读入到 DMA 控制器的某个通道 FIFO 中，然后 DMA 控制器再发出要访问的存储器地址及写控制器信号，将 FIFO 中的数据写入相应的存储器地址。DMA 控制器重复上述操作直到规定的传输内容完成，并通过中断通知 CPU 本次传输完成。

直接存储器访问方式 I/O 除了 CPU 对 DMA 控制器的配置以及相应的中断服务程序外，其数据传输的整个过程完全由硬件实现，没有软件参与，这比通过 CPU 执行 Load/Store 指令进行数据传输的效率要高很多。另外，对于有 Cache 的处理器而言，即使总线被 DMA 占用，CPU 依然可能从指令 Cache 中继续取指运行（直至 Cache 不命中），这也进一步将 CPU 从烦琐的数据传输操作中解放出来。

正因为 DMA 传送方式比一般程序控制传送方式具有数据传送速度高、I/O 响应时间短、CPU 额外开销小的明显优点，所以 DMA 的应用越来越广泛。不仅用于高速 I/O 设备与存储器之间的数据传送，也用于存储器与存储器之间、I/O 设备与 I/O 设备之间的数据传送。通常的应用场合有：Flash 文件系统与主存储器间的数据传输、SD 卡中的数据与主存储器间的数据传输、USB 端口与主存储器间的数据传输、音频接口与主存储器间的数据传输、存储器到存储器间的大块数据复制等。当然，DMA 传送方式的一系列优点通常是以增加系统硬件的复杂性和提高系统的成本为代价的。

事实上，在一款现代 SoC 的设计中，总线或互联架构上的任一款主设备都集成了自己的专用 DMA 通道，这个专用 DMA 通道负责在该主设备与主设备之间进行数据传输，比如 LCD 控制器的主设备接口将会按照配置信息独立地发起从主存储器特定地址（也就是帧缓冲的首地址）到 LCD 控制器的数据传输，将显存中的内容显示到 LCD 屏上。而 CPU 的主设备接口将发起特定传输装载数据 Cache 和指令 Cache（我们称其为 Cache 的行填充）。本节所介绍的是通用 DMA 控制器，而其他主设备的访存接口将在后续章节中分别讨论。

2.6.1 scatter-gather DMA

scatter-gather DMA 方式是与 Block DMA 方式相对应的一种 DMA 方式。在 DMA 传输数据的过程中，一般要求源物理地址和目标物理地址必须是连续的。但是在某些计算机体系中，被传输内容的存储器地址在物理上不一定是连续的，所以 DMA 传输要分成多次完成。如果在传输完一块物理上连续的数据后引起一次中断，然后再由主机进行下一块物理上连续的数据传输，那么这种方式就是 Block DMA 方式。scatter-gather DMA 方式则不同，它使用一个链表描述物理上不连续的存储空间，然后把链表首地址告诉 DMA 主设备。DMA 主设备在传输完一块物理连续的数据后，不用发起中断，而是根据链表来传输下一块物理上连续的数据，直到传输完毕后再发起一次中断。很显然，scatter-gather DMA 方式比 Block DMA 方式效率高。

scatter-gather DMA 方式也称为链式传输方式，DMA 的通道参数（源基地址、目标基地址、数据传输量、下一次描述符指针及其他参数，一组参数称为一个描述符）放在 DMA 外的存储设备内，当前 DMA 过程完成后 DMA 从指针寄存器指向的地址中把新的通道参数读入 DMA 内，按照新的参数要求配置 DMA 并进行新的 DMA 传输过程。由于这些通道参数放在不同的（或相同的）存储区内，上一组通道参数指出下一组通道参数的存放位置，好像链子连接起来一般，故称为链式传输方式。完成 DMA 和这些参数链的初始化后，只要 DMA 接到一个有效的 DMA 请求，DMA 将按照以上的操作反复进行“一个 DMA 过程-取通道参数”，直到最后一个描述符要求的 DMA 过程传输完成。在链式

方式下, DMA 判断全部描述符完成的方法通常是把最后一个描述符的下一个描述符指针置全零, 规定描述符指针全零表示通道参数链的结束。

2.6.2 SEP4020 芯片中的 DMA 控制器

SEP4020 嵌入式微处理器的 DMA 控制器提供如下功能:

- ① 6 个 DMA 通道, 均可支持双向传输。
- ② 提供 Single DMA 和 Burst DMA 请求模式。Burst 传输的尺寸可编程配置。
- ③ 提供存储器到外设、外设到存储器、存储器到存储器的数据传输。
- ④ 支持芯片外部 DMA 请求和响应。
- ⑤ 硬件配置 DMA 通道优先级。
- ⑥ 每个通道对应一组独立的编程寄存器。
 - 可编程的目标地址寄存器、源地址寄存器;
 - 可编程的传输类型 (存储器到存储器、存储器到外设、外设到存储器);
 - 可编程的 AMBA Burst 传输尺寸;
 - 可编程的通道使能;
 - 可编程的 DMA Burst 尺寸。
- ⑦ 地址产生可配置为递增或非递增, 不支持卷址。
- ⑧ 软件配置 Burst 请求支持穿越 1KB 地址边界。
- ⑨ 支持 scatter-gather 的地址生成方式。
- ⑩ 支持基于链表配置的 DMAC。
- ⑪ 支持锁通道功能。
- ⑫ 6 通道共用一个 16×32 位 FIFO。
- ⑬ 自动对数据进行打包、解包以适合 FIFO 宽度。
- ⑭ 可配置的传输控制方: 外设或者 DMA 控制器本身。
- ⑮ 两个中断请求: DMA 错误和 DMA 传输完成中断请求。

SEP4020 的 DMAC 模块介于 AHB 和 APB 两条内部总线之间, 具有总线桥 (BRIDGE) 的功能 (见图 2-36)。当 AHB 总线需要访问 APB 上的资源时, 其向 DMAC 发出请求 (BRIDGE REQUEST), DMAC 把 APB 和 AHB 总线桥接在一起, 从而使得在 AHB 上的 Master 设备可以访问 APB 总线的设备; 反之, 当使能 DMAC 传输时, DMAC 可以申请占用 AHB 总线, 同时 DMAC 直接控制 APB 上的数据、地址和控制信号。通过 DMAC 模块, 可以快速实现源 (SOURCE) 和目的 (DESTINATION) 之间数据的传送, 减轻 CPU 的负担。DMAC 包括以下的功能模块: AHB Slave 接口、AHB Mater 接口、APB 接口, 以及 DMAC 接口, 中断和内部 FIFO。具体组成结构见图 2-28。

AHB Slave Interface 用于 CPU 访问 DMAC 的寄存器。通过访问这些寄存器可以对 DMAC 进行配置和监控。所有的 AHB Slave 编程总线宽度都是 32 位。

AHB Master 接口 (AHB Master Interface): DMAC 模块本身而言也是一个 AHB 总线上的 Master 设备。通过 AHB Master 接口模块, DMAC 和 AHB 总线相连接。在进行 DMA 操作时, DMAC 可以申请 AHB 总线的使用权从而可以直接访问 AHB 上的资源。默认的 AHB 总线宽度是 32 位总线。当使用 DMA 传输时, 源和目的数据可以是不同的宽度, 可以等于或小于总线的物理宽度。DMAC 会对数据进行适当的打包和解包以满足 AHB 总线宽度。除非有特殊声明, 接口必须按照 AMBA 规范 (2.0) 连接。

APB 接口 (APB Interface): 当进行 DMA 操作时, DMAC 模块替代 APB 直接产生地址、数据和控制信号, 从而可以直接访问 APB 上的资源。

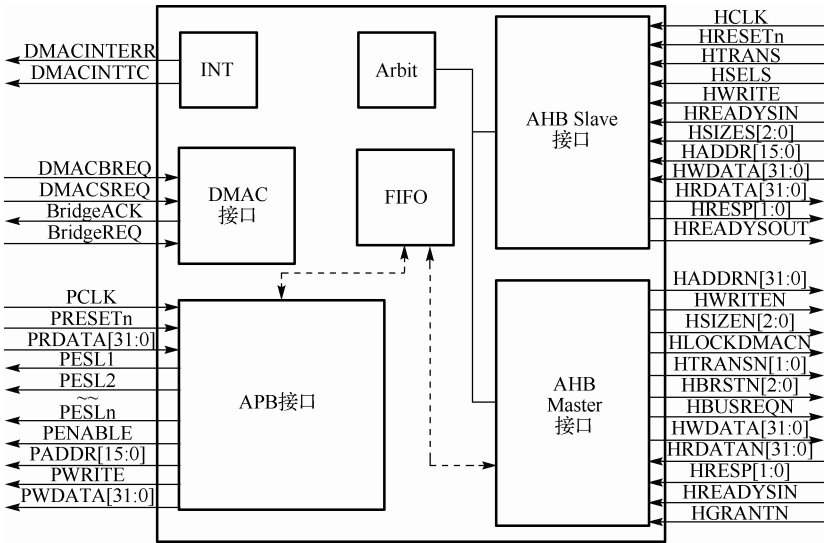


图 2-28 SEP4020 的 DMAC 结构框图

中断请求（Interrupt Request）用于产生中断。DMAC 提供两种中断（DMACINTERR 和 DMACINTTC）与系统的中断控制器（INTC）相连。两种中断都是由多个可屏蔽的中断源线或而成。在发生 AHB 错误或传输完成时将发出中断请求。

DMA 请求和响应接口（DMA Interface）包括两组信号：一组信号来自 APB 总线；另一组信号来自 APB 上的功能模块（如 SPI 等等），如图 2-29 所示。各信号意义如下：

- BridgeREQ：表示当前 APB 向 DMAC 申请使用 APB 总线上的资源。
- BridgeACK：表示当前 DMAC 同意 APB 的请求。
- DMACBREQ：表示 APB 上的外设申请 Burst DMA 请求。
- DMACSREQ：表示 APB 上的外设申请 Single DMA 请求。

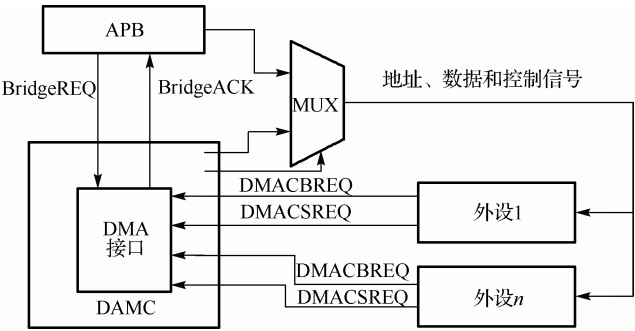


图 2-29 DMAC 请求和响应接口示意图

当 APB 希望访问 APB 总线上的资源时，其使 BridgeREQ 信号有效以提出申请。若 DMAC 没有占用 APB 总线则反馈 BridgeACK 信号作为应答，这样 CPU 就可以通过 APB 访问 APB 总线上的设备了。当外设需要申请一次 DMA 操作时，其使能 DMACBREQ 或 DMACSREQ 信号并且一直保持该信号直到 DMAC 对提出请求的外设进行操作。

在 SEP4020 处理器中的 DMA 设计之所以要将 DMA 控制器与 AHB 到 APB 总线桥集成到一起，是因为在大多数情况下每次 DMA 交易可以减少一次 DMA 控制器申请 AHB 总线。如果 DMA 控制器

作为一个独立的 AHB 主设备（见图 2-30(a)），每次 DMA 交易，DMA 控制器首先要申请 AHB 总线，发起将 APB 低速设备的数据读取到 DMA 控制器内部的 FIFO 中，然后 DMA 控制器将再次申请 AHB 总线，将 FIFO 中的数据写入到外部存储器。而采用 SEP4020 的 DMA 架构（见图 2-30(b)），DMA 发起将低速设备数据读入 FIFO 的操作将不用申请 AHB 总线，只是在第二步将 FIFO 数据写入外部存储器时才需要申请 AHB 总线。

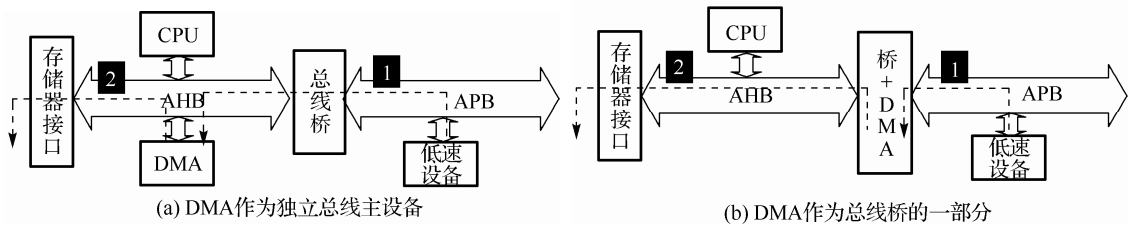


图 2-30 DMAC 作为独立总线主设备和作为总线桥的一部分

SEP4020 DMAC 通道及优先级：DMAC 共有 6 个可配置的硬件通道。每一个 DMA 操作配置一个硬件通道。这样可以灵活地为请求 DMA 的设备分配硬件通道。通道的优先级固定。DMAC 6 个通道描述见表 2-3。

表 2-3 DMAC 通道描述

通道数	优先级	响应概率 ¹
通道 0	优先级（一）	3/6
通道 1		
通道 2	优先级（二）	2/6
通道 3		
通道 4	优先级（三）	1/6
通道 5		

注释 1：DMAC 有 6 个通道，分为 3 组优先级别。响应概率越高优先级越高。通道 0 和 1 的优先级最高；通道 4 和 5 的优先级最低。同一组中的两个通道有相同的优先级。

SEP4020 DMA 控制器地址产生和 Burst 操作：地址产生可以是递增（incrementing）或非递增（non-incrementing）的，但不支持卷址（address wrapping）。当 DMAC 为控制方时，每个通道可以定义的最大传输尺寸为 4095 个数据（数据尺寸可以是 BYTE、HALFWORD 或者 WORD）。Burst 操作的长度可以是 4、8、16 或者是单个数据。值得特别指出的是 Burst 操作不能超越 1KB 的地址边界^①。因此，用户需要小心使用固定长度（4、8 或 16）的地址递增的 Burst 操作防止其穿越了 1KB 的地址边界。它支持使用单个数据长度的（长度为 1）地址递增的 Burst 操作。

假设数据传输宽度为 Word，起始地址为 0x1fff_03f0。设置地址递增。如果设置 Burst 长度为 8 或者 16 则会产生超越 1KB 地址边界的错误。在这种情况下，可以设置 Burst 长度为 4 或者 1。长度为 4 时，Burst 操作正好结束于边界处；长度为 1 时，单个数据长度 Burst 操作可以跨越边界。

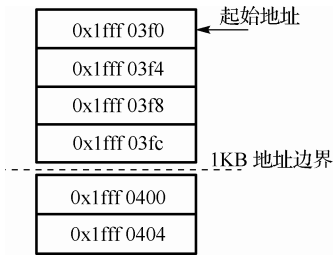


图 2-31 地址产生和 Burst 操作

① 这个限制是由于 AMBA 2 总线标准决定的，具体可参见 AMBA 2 总线协议。

2.6.3 DMAC 驱动

在介绍 DMAC 驱动的编写之前，先来看看 SEP4020 的 DMAC 控制器模块中，如果需要配置某一个 DMA 通道进行 DMA 传输时需要对哪些寄存器进行配置：

- ① DMA 通道 x 源地址寄存器。
- ② DMA 通道 x 目的地址寄存器。
- ③ DMA 通道 x 控制寄存器。
 - Burst 传输尺寸：byte、halfword、word；
 - Burst 传输大小：1、4、8、16；
 - 源地址增加模式（地址增加或非增）；
 - 目标地址增加模式（地址增加或非增）。
- ④ DMA 通道 x 配置寄存器。
 - 传输方向：存储器到外设、外设到存储器，存储器到存储器；
 - 中断屏蔽；
 - 使能 DMAC 通道：DMAC 每个通道的配置寄存器 0 位是该通道的使能位，当被设成 1 时，通道被打开。

如果要进行存储器到外设的 DMA 传送，驱动程序流程图如图 2-32 所示。

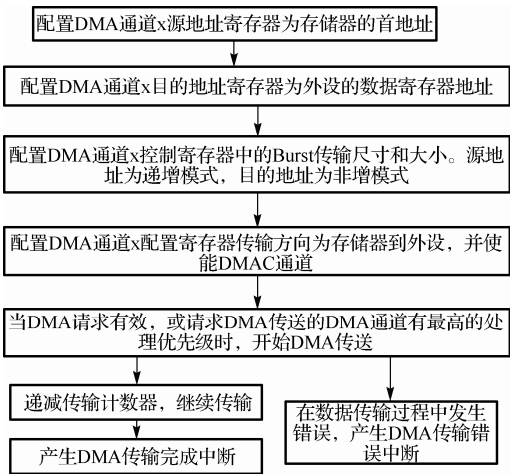


图 2-32 存储器到外设 DMA 传送驱动程序流程图

2.7 外设接口控制器

2.7.1 高速通信接口控制器

根据所面向应用的不同，现代 SoC 往往在芯片内继承了可以用于高速数据传输的通信接口控制器，比如 PCI 总线接口、以太网控制器、USB 总线接口等。USB 高速通信接口目前广泛应用于嵌入式系统中，智能手机、平板电脑等终端设备可以通过 USB 线和 PC 之间进行数据传输。嵌入式系统 SoC 中通常集成有 USB 接口控制器。

关于 USB 接口和以太网控制器的介绍、控制器的设计以及相应驱动编写等详见第 6 章 6.2 节。

2.7.2 低速通信接口控制器

在嵌入式系统中,还存在大量低速外设,像传感器、低速 Flash 存储器、字符型液晶显示模块等。这些外设只需要简单的、引脚数较少的低速接口,嵌入式系统 SoC 通常集成有相应的接口控制器。

多数低速接口采用串行传输,以减少使用的信号线数量(也就是芯片的引脚数量)。处理器通过串行传输控制器与外设进行串行传输。串行传输控制器将处理器传送过来的要发送的并行数据转换为串行数据流输出,并将外设来的串行数据转换为并行数据,供处理器使用。处理器把准备发送的数据写入串行传输控制器的发送数据寄存器中;相应的处理器可以从串行传输控制器的接收数据寄存器依次读出接收到的数据。发送数据寄存器与接收数据寄存器通常分别是一组 FIFO 寄存器,同时设计有基于 FIFO 使用状态的中断触发机制。控制器发送 FIFO 状态为空或接近空时,控制器可以通过中断提醒处理器装载下一组要发送的数据;控制器接收 FIFO 状态为满或接近满时,控制器可以通过中断提醒处理器读出缓存的数据。通过这种机制,可以有效减少处理器对于传输控制器的状态轮询操作,同时还可以有效提升传输效率。另外,部分传输控制器还支持 DMA 传输机制,大幅减轻了处理器进行数据传输时工作负载。

关于 I2C、SPI、UART 等低速通信接口的介绍以及相应控制器的设计和驱动编写等内容详见第 6 章 6.1 节。

2.7.3 人机界面控制器

人机交互(Human-Machine Interaction)是指人与机器的交互,本质上是人与计算机的交互。人机交互,即用户与计算机之间的双向通信,通过一定的符号和动作来实现,如击键、移动鼠标、显示屏幕上的符号/图形等。这个过程包括几个子过程:识别交互对象、理解交互对象、把握对象情态、信息适应与反馈等。

人机界面(Human-Machine Interface, HMI)则是用户与计算机之间的通信媒体或手段,它的物化体现在相关硬件及支持软件,如带有鼠标的图形显示终端等。

人机交互是人与计算机作用关系/状况的一种描述;人机界面是人与计算机发生交互关系的具体表达形式。交互是实现信息传达的情境刻画,而界面是实现交互的手段。交互是内容/灵魂,界面是形式/肉体,两者的最终目的都是为了解决人机关系、满足人的需求。

凡参与人机交互的领域都存在着人机界面。在嵌入式系统中,人机界面被大量使用。人机界面可以简单地地区分为输入与输出。输入指的是由人来进行设备的操作,下达指令控制设备运行;而输出指的是由设备发出来的通知,如状态信息、操作说明等。好的人机界面会帮助用户更简单、更迅速地正确操作设备,发挥设备的最大效率。对于智能手持终端类设备而言,最常见的人机界面设备包括:液晶显示、触摸屏以及相应的音频接口(声音的播放与录入)。

关于液晶显示、音频接口、触摸屏原理的介绍、SoC 中这些人机界面控制器的设计、这些人机界面控制器的驱动等内容详见第 6 章 6.3 节。

2.8 案例: SoC 架构设计

2.8.1 S3C44B0X

三星公司推出的 S3C44B0X 是一款基于 ARM7TDMI 内核的 16/32 位 RISC 处理器,扩展了一系列完整的通用外设模块,使系统的费用降至最低,降低了硬件开发的难度。S3C44B0X 集成了丰富的

内部模块，包括内部 SRAM、SDRAM 控制器、LCD 控制器、4 通道 DMA、6 通道定时器、2 通道 UART、I/O 端口、RTC、8 通道 10 位 ADC、I²C 接口控制器、I²S 接口控制器等。

S3C44B0X 是广泛使用的基于 ARM7TDMI 内核的 SoC。在当时，该芯片功能强大，为早期的手持设备和一般类型应用提供了高性能、高性价比的嵌入式处理器解决方案。同时，它出色的低功耗设计特别适用于对功耗敏感的应用。图 2-33 为 S3C44B0X 的结构框图。

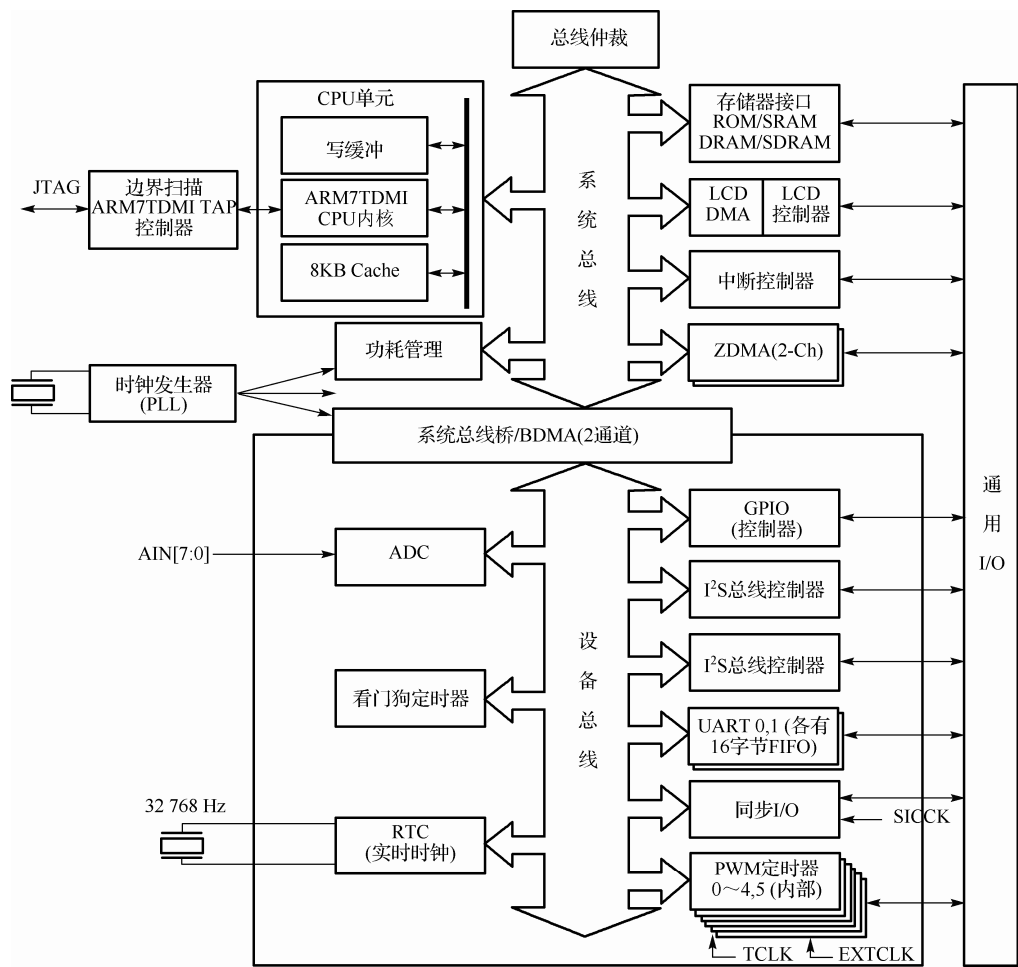


图 2-33 S3C44B0X 的结构框图

2.8.2 S3C6410

三星公司推出的 S3C6410 基于 ARM1176JZF-S 内核，采用 64/32 位的内部总线架构，融合了 AXI、AHB、APB 总线。还有很多强大的硬件加速器，用于视频处理、音频处理、2D/3D 图形处理、显示处理等。S3C6410 包括优化的外部存储器接口，该接口能满足在高端通信服务中的数据带宽要求。该接口支持 DRAM、Flash、ROM、SRAM 等类型的外部存储器。为了降低整个系统的成本，S3C6410 也集成了多种外设模块：Camera 接口控制器、LCD 控制器、4 通道的 UART、32 通道 DMA、4 通道定时器、通用 I/O 端口、I²S 接口控制器、I²C 接口控制器、USB（Host 及 OTG）控制器、SD/MMC 卡接口控制器等。

S3C6410 在当时是一款高性能的手持移动终端通用处理器，可以满足手持移动终端丰富的多媒体应用需求。图 2-34 为 S3C6410X 的结构框图。

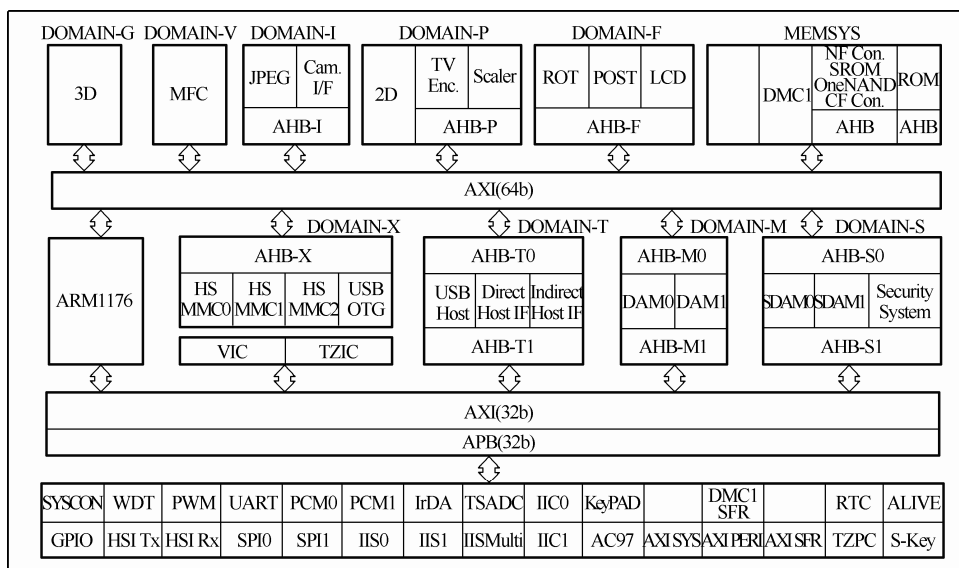


图 2-34 S3C6410X 的结构框图

2.8.3 OMAP3530

TI 公司推出的 OMAP3530 是专门为智能手机、GPS 系统和平板电脑等低功耗便携式应用而设计的。OMAP3530 集成了 ARM Cortex-A8 内核、TMS320C64 DSP 内核、图形引擎、视频加速器以及丰富的多媒体外设，其中 Cortex-A8 内核与 ARM11 系列相比，可获得多至 3 倍的性能增益。OMAP3530 可用于流媒体、2D/3D 游戏、视频会议、高清静态图像等应用。图 2-35 为 OMAP3530 的结构框图。

2.8.4 SEP4020

SEP4020 由东南大学国家专用集成电路系统工程技术研究中心设计，使用 0.18μm 标准 CMOS 的工艺设计，内嵌 ASIX CORE (32 位 RISC 内核，兼容 ARM720T，带 8KB 指令数据 Cache 和全功能 MMU)，采用冯·诺依曼结构，SEP4020 芯片中集成的各种功能包括：

- 8/16 位 SRAM/NOR FLASH 接口，16 位 SDRAM 接口。
- 硬件 NAND FLASH 控制器，支持 NAND FLASH 自启动，支持软件/硬件 ECC 校验。
- 10M/100M 自适应以太网 MAC，支持 RMII 接口。
- 64KB 高速片上 SRAM。
- USB1.1 Device，全速 12Mbps。
- 支持 I2S 音频接口。
- 支持 MMC/SD 卡。
- LCD 控制器，支持 640×480×16 位 TFT 彩屏和 STN 黑白、灰度屏。
- RTC，支持日历功能/WatchDos，支持后备电源。
- 10 通道 TIMER，支持捕获、外部时钟驱动和 MATCH OUT。
- 4 通道 PWM，支持高速 GPIO。
- 4 通道 UART，均支持红外。
- 2 通道 SSI，支持 SPI 和 Microwire 协议。
- 2 通道 SmardCard 接口，兼容 ISO7816 协议。

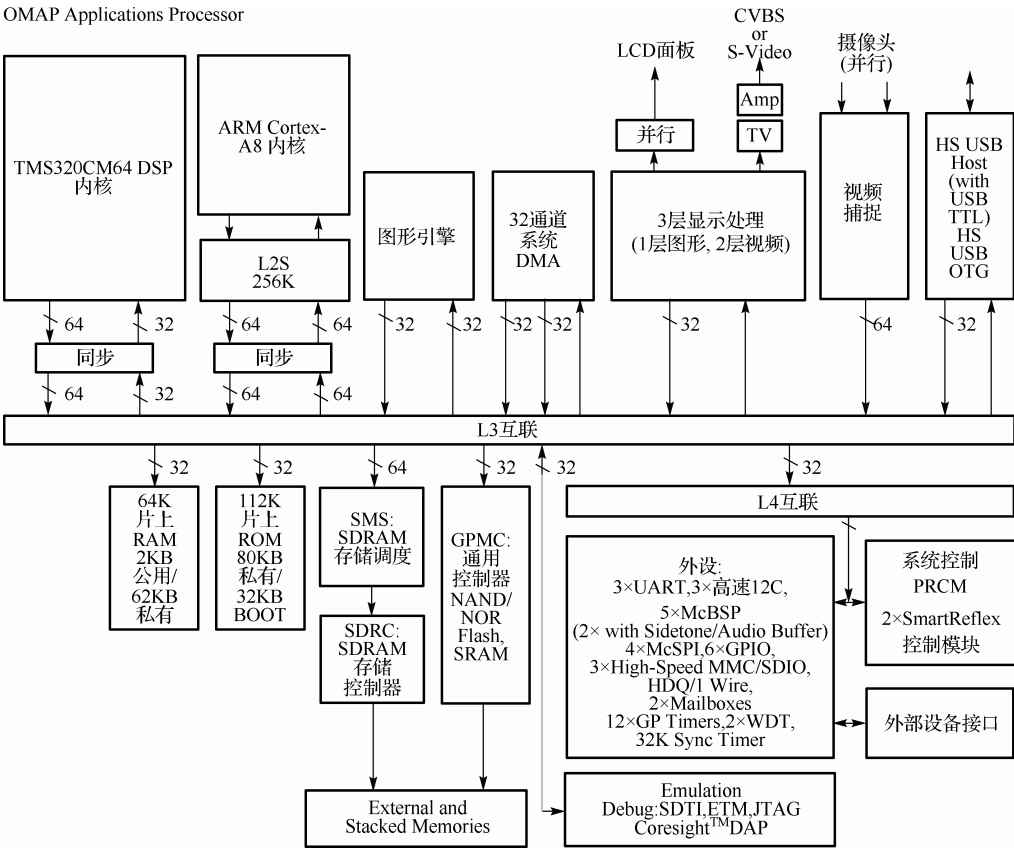


图 2-35 OMAP3530 的结构框图

- 支持最多 97 个 GPIO，14 个外部中断。
- 支持链表 DMA 传输和外部 DMA 传输。
- 片上 DPLL，支持多种功耗模式：IDLE、SLOW、NORMAL、SLEEP。

SEP4020 处理器架构图如图 2-36 所示。

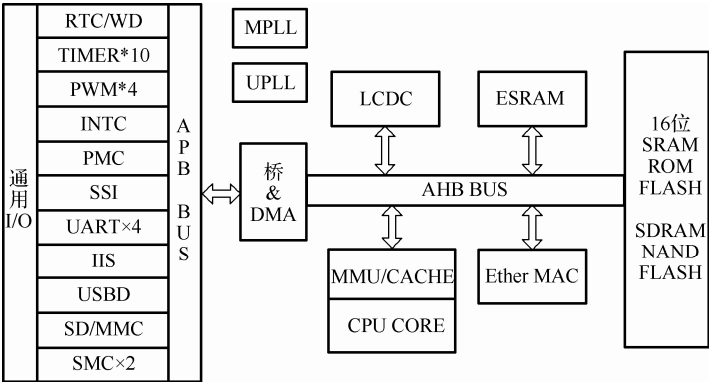


图 2-36 SEP4020 处理器框架构图

2.8.5 SEP6200

SEP6200 由东南大学国家专用集成电路系统工程技术研究设计中心设计，定位于手持视频播放设备、

卫星导航产品的高性能应用处理器,采用 TSMC65nm 工艺。SEP6200 主要分为 5 个部分:系统与时钟控制、外设与接口、多媒体系统、GPS 系统和存储系统,如图 2-37 所示。SEP6200 主要有以下特性:

- UniCore-2 内核^①,主频最高 500MHz;
- 多总线架构,高总线利用率;
- 支持 DDR2/SRAM/NOR/NAND 存储器接口;
- 支持 H.264/MPEG4/DivX/VC1/RV 等主流视频格式的 1080P 解码;
- 支持外接 AUDIO CODEC 芯片实现音频输入/输出;
- 支持外接 HDMI 芯片实现 HDMI 视频输出;
- 支持 24 位色 LCD 输出,可支持 4 层 Overlay;
- 内嵌 USB OTG2.0 控制器,支持外接 PHY 实现 USB2.0 传输;
- 支持 SDIO 接口和 SDHC 存储卡;
- 高速 SPI 支持 WiFi 接口;
- 支持年月日时分秒实时时钟;
- 支持 I²C 接口;
- 支持 GPS 定位;
- 内嵌电源管理,可动态调整频率。

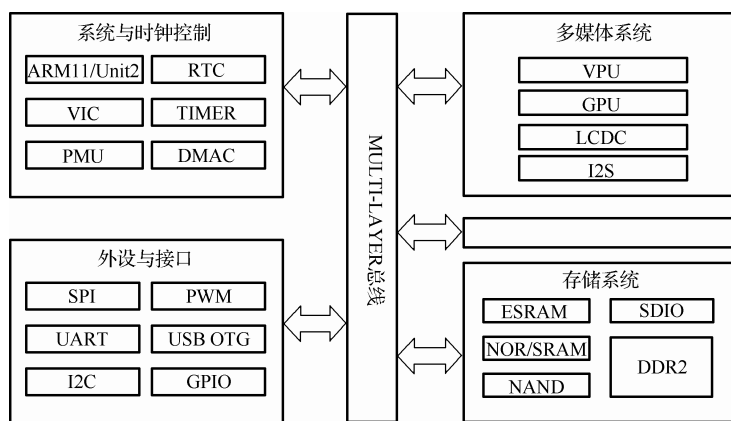


图 2-37 SEP6200 结构框图

SEP6200A 是东南大学国家专用集成电路系统工程技术研究中心设计实现的一个移动智能终端 SoC 芯片。令人遗憾的是,由于设计经验不足使得第一版设计 SEP6200A 的架构存在缺陷,导致 SEP6200A 的性能没有得到正常发挥。SEP6200A 的系统架构如图 2-38 所示。用于连接主处理器 UniCore 和 DDR 存储控制器的总线 BUS1,由于同时还连接了其他挂接了多个设备的总线,比如 BUS2、BUS3 和 BUS4,使得该总线非常容易产生阻塞,导致 UniCore 的访存速度较低,影响系统整体性能。

为提升 SEP6200A 的性能,在 SEP6200A 的基础上,工程中心设计实现了改进版本 SEP6200B。SEP6200B 的系统架构如图 2-39 所示。SEP6200B 的系统架构与 SEP6200A 最大的区别在于使用高速总线 BUS0,将主处理器 UniCore 到 DDR 存储器的数据传输与到系统其他设备的传输分离开来,提升了 UniCore 的访存速度,进而提升了系统整体性能。性能提升结果如图 2-40 所示,改进后的架构使得 Linux 内核启动时间从 2.42s 降低到 1.11s,降低了 54%;GPU 测试场景的渲染时间从 0.23s 降低到 0.12s,降低了 48%。系统性能得到有效的全面提升。

^① UniCore-2 CPU 内核是由北京大学研发的 32 位 RISC 处理器内核,关于该 CPU 的介绍可以参阅第 4 章 4.3.3 节。

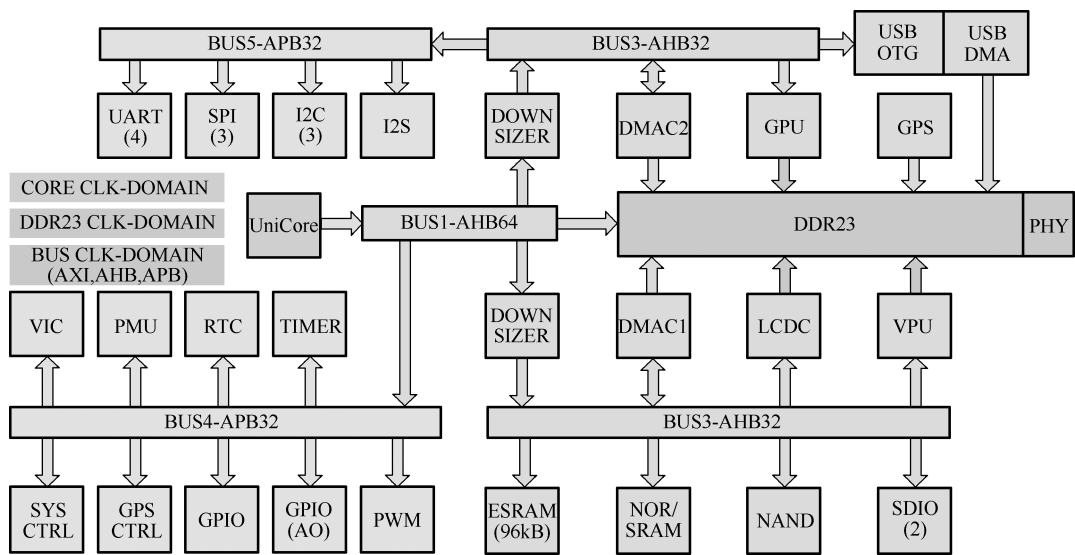


图 2-38 SEP6200A 结构框图

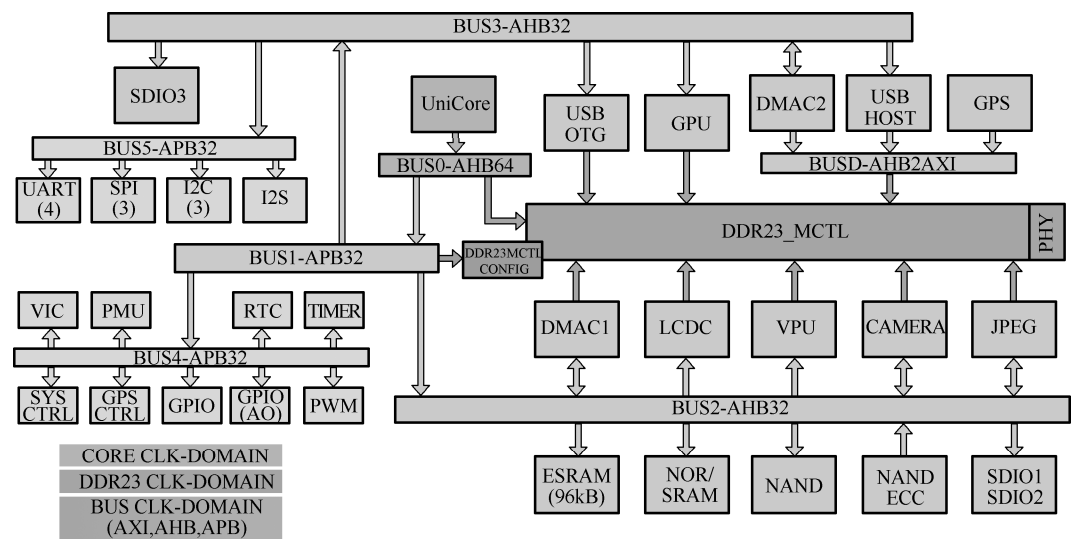


图 2-39 SEP6200B 结构框图

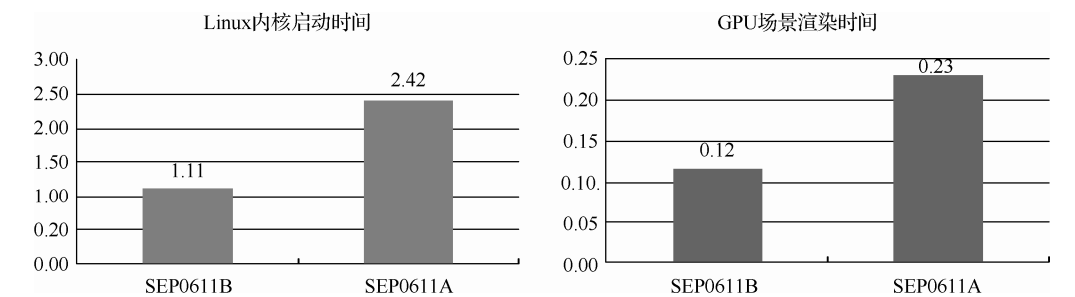


图 2-40 SEP6200A 与 SEP6200B 性能对比^①

① 图中 SEP0611A 和 SEP0611B 分别是 SEP6200A 和 SEP6200B 两款处理器的内部代号。

思考题

1. 比较 PC 架构与一般 SoC 架构的异同, 请思考为什么 PC 架构没有采用 SoC 芯片通常采用的架构, 将整个系统集成到同一硅片上, 而是采用北桥和南桥芯片进行扩展?

2. 研究 AHB 协议后我们会发现在 AHB 总线的标准中并没有像 PC 总线(如 PCI)那样定义中断信号, 讨论一下为什么?

3. DMA 控制器在发起数据传输的时候将申请占用系统总线, 并访问存储器。因此对于没有采用 Cache 的处理器而言, 总线被 DMA 占用期间 CPU 是无法取指的, 也就是说在此期间 CPU 只能等待。基于以上分析, 有人认为在这样的系统中采用 DMA 机制, 相比于通过 CPU 执行软件进行数据搬运并不能提高系统数据的传输效率。请思考这样的结论是否正确? 为什么?

4. 请读者自己设计 SoC 的硬件架构框架。功能需求: 用于视频数据采集用摄像头(如手机用的针孔摄像头)拍摄视频, 分辨率大约在 320×240 ppi; 把视频数据流进行简单加密, 然后把加密后的视频数据流存放到 microSD 卡中; 用专用的 PC 软件把 microSD 卡中的文件进行解密、解码后在 PC 上播放出来; 要求可以配置每秒钟拍几帧图像, 可配置为 1 秒、2 秒或 3 秒拍一张, 最多配置为每秒拍 15 张。

扩展阅读

- [1] William Stallings. 计算机组织与体系结构. 第 7 版. 张昆藏等译. 北京: 清华大学出版社, 2006.
- [2] Randal E. Bryant, David O'Hallaron. 深入理解计算机系统. 北京: 中国电力出版社, 2010.
- [3] Hennessy J, Patterson D. Computer Architecture: A Quantitative Approach. 4th edition. Morgan Kaufmann Publishers, Menlo Park, CA, 2006.
- [4] ARM Ltd. AMBA Specification(REV2.0). ARM Ltd, 1999.
- [5] ARM Ltd. AHB Transaction Modeling. ARM Ltd, 2006.
- [6] ARM Ltd. AMBA AXI and ACE Protocol Specification. ARM Ltd, 2003.
- [7] Lee, Hyung Gyu, Naehyuck Chang, Umit Y. Ogras, and Radu Marculescu. On-chip communication architecture exploration: A quantitative evaluation of point-to-point, bus, and network-on-chip approaches. ACM Transactions on Design Automation of Electronic Systems (TODAES) 12, no. 3 (2007): 23.
- [8] Marculescu, Radu, Umit Y. Ogras, Li-Shiuan Peh, Natalie Enright Jerger, and Yatin Hoskote. Outstanding research problems in NoC design: system, microarchitecture, and circuit perspectives. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 28, no. 1 (2009): 3-21.
- [9] Michael Keating, Pierre Bricaud, Reuse Methodology Manual for System-on-a-Chip Design. Springer, 2002.
- [10] SEP4020 嵌入式微处理器用户手册. 南京: 东南大学国家 ASIC 工程中心, 2008.

第 3 章 嵌入式系统的开发和调试

3.1 嵌入式系统的一般开发过程

嵌入式系统的调试过程最常见的方式是交叉编译的调试方式，也就是嵌入式软件源代码的编辑、编译、连接等过程都在主机（通常是 PC）上完成，程序员通过运行在主机上的调试器将编译连接生成的镜像文件下载到所调试的嵌入式系统开发板中（通常称其为目标系统或目标板），并通过调试器监控程序在目标板上的运行过程。嵌入式系统的一般开发流程如图 3-1 所示。

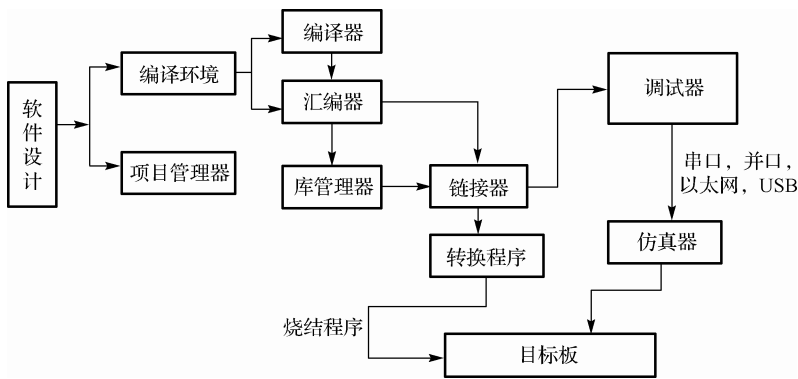


图 3-1 嵌入式系统的开发流程

图 3-1 中，编辑器负责源代码的录入以及源码文件的管理。项目 managers 的主要作用是维护一个软件项目的编译环境，采用项目 managers 的最大好处是把程序员从烦琐的 **Make** 文件编写中解放出来。编译器、汇编器、库管理器、链接器属于传统意义上的工具链，它们负责将用户的源代码分别转换为机器码，并将多个机器码文件拼装成为统一编址的输出文件。调试器的作用是负责将链接器输出的文件装载到目标系统中，并监控程序的运行，实现代码的调试。

由于通常情况下链接器输出的文件并不是纯粹的二进制机器码流，而是包含了大量用于调试和程序加载（Loading）信息的特定格式（比如标准的 ELF 格式），这种输出文件并不能直接在目标板上运行。在调试阶段可以通过调试器读取相关的加载信息，并通过仿真器（Emulator）将输出文件中所包含的二进制码流加载到目标板相应的内存地址。在实际的产品发布阶段，对于有操作系统支持的系统（通常具有相应的文件系统和应用程序加载器），链接器输出的文件可以作为文件存放在目标板的文件系统中（比如 **Nand Flash** 文件系统），当需要运行该程序时，操作系统的加载器将从文件系统中读取该输出文件，提取其中的二进制码流，并根据相应的加载信息将其加载到相应的内存地址，最终由操作系统调用该地址运行该程序；对于没有操作系统或操作系统不支持应用程序加载的系统而言，通常需要通过专门的转换程序将链接输出文件转换为纯的二进制码流，并通过相应的烧写程序将二进制码流直接烧写到目标系统的 **Flash** 存储器中（如图 3-1 所示），由于此类系统往往是将 OS 与应用程序统一链接在一个镜像中，因此 OS 可以像调用函数那样直接运行用户的应用程序。关于这部分内容有兴趣的读者可以参阅本章扩展阅读[1]。

本节将分别介绍交叉编译、链接和调试 3 个主要过程。

3.1.1 交叉编译

所谓交叉编译（Cross Compiling）是指编译、链接源代码的计算机与执行该代码的计算机是不同的计算机（通常也是不同的处理器架构）。在嵌入式系统的开发过程中通常采用 PC 作为主机，执行交叉编译、链接等工作；目标机指运行嵌入式软件的硬件平台，也就是开发人员所需要调试的系统。图 3-2 以文件的形式进一步描述了嵌入式系统的交叉编译过程。对 C 语言源文件利用交叉 C 编译器生成相应的汇编文件（扩展名一般是.s），然后编译器调用汇编器将相关的汇编文件汇编为目标文件（一般扩展名为.o）。人们往往误解编译器直接输出的是目标文件，其实编译器首先将 C 文件编译为汇编文件，然后编译器再调用汇编器生成目标文件。汇编文件（包括汇编语言源文件和 C 语言编译后生成的汇编文件）经过汇编器生成*.o 文件，若干个*.o 文件需要经过链接器生成与目标系统存储器地址相关的链接输出文件 file.out（不同的链接器输出的格式可能不一样），此文件包含很多调试信息，例如全局符号表、C 语句所对应的汇编语句等。调试信息的格式可以是厂商自己定义的，也可以是遵循标准的（如 IEEE695）。软件开发人员还可以利用 Liber 工具将若干个目标文件合并成为一个库文件。一般而言，一个库文件提供了一组功能相对独立的工具函数集，比如操作系统库、标准 C 函数库、手写识别库等。用户在使用链接工具时，可以将所需要的库文件一起链接到生成的内存镜像文件 file.out 中（这个过程被称为静态链接，现代操作系统还支持在程序运行过程中动态加载并链接需要的库文件，这被称为动态链接）。在以上流程中，这些软件工具还会生成一些辅助性文件，比如编译器在编译一个 C 文件时，会生成该文件的列表文件（*.lst，不同编译器输出的扩展名可能不一样），该文件采用纯文本的方式将 C 文件的语句翻译成为一组相应的汇编语句；链接器除了生成内存镜像文件外，一般还生成一个全局符号表文件（*.xrf，不同链接器输出的文件扩展名可能不一样），开发人员可以利用该文件查看整个镜像文件中的任意变量、任意函数在内存中的绝对地址。file.out 文件可以由调试工具加载到目标系统的 SDRAM 中进行调试，也可以通过转换工具转换为二进制文件，再利用烧写工具烧录到目标系统的 Flash 中。

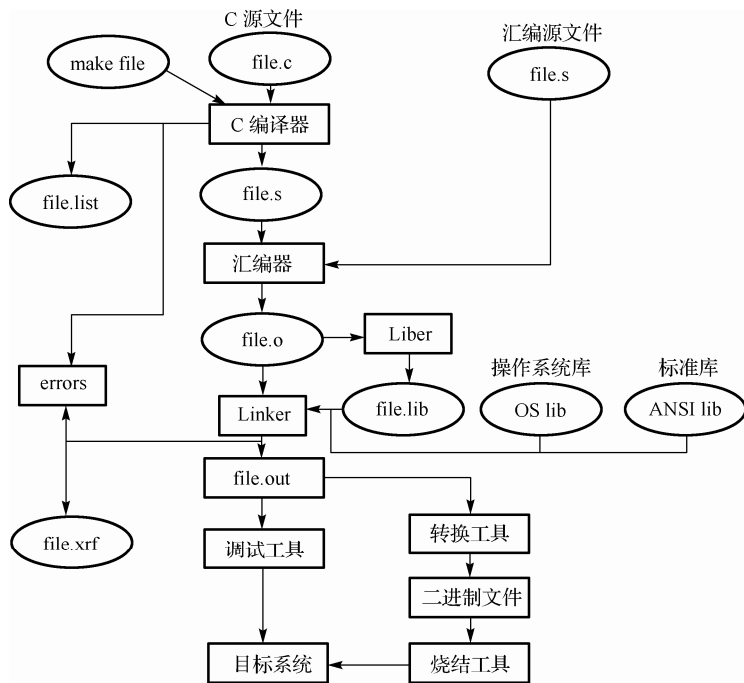


图 3-2 嵌入式系统交叉开发过程

3.1.2 链接

链接的过程也就是从*.o 文件（目标文件）到 file.out 文件的过程。每个目标文件*.o 实际上都是独立的机器码文件，其内部的地址空间是独立于其他目标文件的，所有该文件内符号的编址都是独立进行的，与项目中其他目标文件中的符号编址无关（这是因为 C 语言编译器是以单个 C 文件为编译单位的，当 C 编译器在编译当前 C 文件时，并不知道项目中其他的 C 文件），所以单个的目标文件实际上是无法运行的。每个目标文件的第一条指令都从相同的地址开始存放，一个具体的目标文件 file.o 的结构如图 3-3 所示，图中以 68K 系列微处理器和 ARM 系列微处理器为例说明了目标文件的结构。

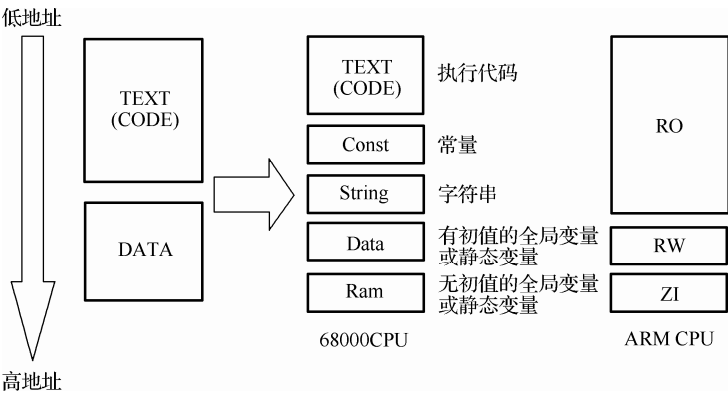


图 3-3 目标文件结构图

程序经过编译后主要由两部分组成：代码和数据。汇编器进行汇编后，得到的数据和代码分开存放，一般低地址端放代码，高地址端放数据，数据又可分为有初值的全局变量或静态变量和无初值的变量两种，系统初始化时会自动将无初值变量初始化为零，如图 3-3 所示，Motorola 的 68K 系列微处理器汇编后段（Segment）分类较多，而 ARM 系列微处理器经汇编后的段只有 3 种，分为只读段 RO（包括代码和常量）、有初值的全局变量 RW（可读可写段）和无初值的变量 ZI（零初始化段）。汇编器生成的*.o 文件将这 3 种段从零地址开始顺序存放。

链接器将编译或汇编生成过的目标文件以及操作系统库文件、标准 C 函数库文件等库文件链接到一起，链接的过程实际上是将各个独立的目标地址空间编排到一个统一的地址空间中，生成一个完整的与实际物理内存相符合的内存映像文件（file.out 文件），图 3-4 是以 68K 为例说明缺省情况下链接器如何将不同的目标文件链接起来。各个不同的目标文件都有各自不同的代码、常量、数据和存储空间，经链接器链接后，所有的代码、数据等文件格式都统一地存放在一个绝对的物理地址中。

3.1.3 调试

由链接器链接后生成与目标系统存储器地址相关的目标文件可以以两种方式在目标板上运行：

- 目标文件送给调试器（Debugger），调试器通过并口、串口、以太网口或 USB 口与一个称为“仿真器”^①的硬件设备通信，并通过该“仿真器”把目标下载到目标板运行并调试。
- 目标文件烧录到目标板上。链接器生成的目标文件通常并不是可以直接烧录到目标板上的二进制格式，此时就需要一个转换程序把目标文件转换成二进制格式，并将纯二进制格式的镜像烧录到目标板上的 ROM 或 Flash 存储器运行。

① 我们将在本章 3.2 节介绍片上在线仿真调试，通过对该节的阅读，读者将知道为什么我们会在这里加上引号。

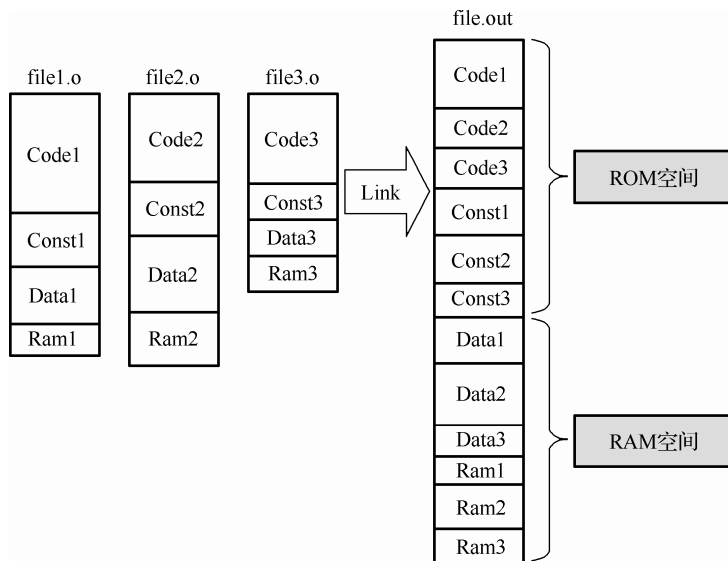


图 3-4 链接器的配置规则（68K）

在嵌入式系统开发的初期，程序还没有定型因而需要不断调试时，通常是把目标文件送给调试器通过仿真器下载到目标板上运行并调试。当目标板上的程序调试成功可以定型了，就可以把目标文件通过转换程序转换为二进制格式烧录到目标板上运行。烧录到目标板上的程序是无法进行仿真调试的。我们将在下一节详细介绍调试的方法。

3.2 调试方式介绍

自计算机产生以来，对程序的调试就是工程设计中不可缺少的一个步骤，并且随着软件设计在工程设计中所占比例的日益增大，软件调试也受到越来越多的关注。任何计算机系统的调试都是一件复杂的任务，调试有两种基本的方法：最简单的方法是使用如逻辑分析仪一类的测试仪器从外部监视系统；更强有力的方法是使用支持单步执行、设置断点等功能的工具从内部观察系统。本章所介绍的调试指的是从内部观察系统的运行情况。

一般的调试系统都应该具有以下几个基本特点：

- 控制程序的执行；
- 检查与改变处理器的状态；
- 检查与改变整个系统的状态。

其中，对程序执行的控制包括对感兴趣的数据进行观察、对感兴趣的指令设置断点以及对程序的单步执行。值得大家关注的是现在大部分调试器都支持基于 C 语言的源级调试，在具体程序的单步执行过程中，一条 C 语句可能对应若干条汇编语句，所以在具体的调试过程中，尽量选择最低的编译器优化选项，这样可以比较直观地观察到整个程序的运行状态。ARM 公司的 RVDS 开发套件提供的 AXD 调试器还提供了通过查看编译后的符号表实现对变量的监测功能，它不仅提供了断点（breakpoint）功能，还实现了观察点（watchpoint）功能。提到断点大家可能都比较熟悉，因为在 VC、Turbo C 中断点都有所应用，它主要是针对程序而言的；而观察点主要是针对数据而言的，在 C 程序中，任何一个变量尤其是全局变量都对应着一个存储地址，所以总是可以用一个指针来表述的，通过 watchpoint 这个功能，可以将任何一个数据对应的地址设为观察点，当观察点的数据发生改变时，程序就会停止执

行，这在软件开发中是相当重要的，因为它可以监视数据的改变，防止对数据的误操作导致程序的不可用，这在查询缓冲区溢出、内存泄漏等错误时是非常有效的工具。

检查与改变处理器的状态是指在调试器中可以对处理器内部的寄存器进行读写操作，可以查看并修改这些寄存器的值。通常大多数的调试器都支持在程序被暂停的情况下，将当前处理器（这里所说的处理器是指 CPU）所有寄存器的值更新到调试器的界面，这样程序员就可以知道当前处理器内部的情况。在断点情况下，调试人员甚至可以动态地更新寄存器的值，并在完成更新后让程序接着运行。

检查与改变系统的状态包括对系统存储器的访问（包括读和写）、下载代码到系统存储器中、把系统存储器的内容提取出来保存到文件中等。调试器提供对系统存储器的访问，可以查看并改变特定存储器地址的内容，这对系统的调试是非常重要的，可以实现对局部变量和全局变量的变化进行跟踪。RVDS 开发套件中的 AXD 调试器实现了将程序下载到目标板上的功能，还可以将目标板存储器的内容保存到 PC 的文件中，这在 mp3 测试、信噪比分析等应用中是非常重要的。另外，由于大多数嵌入式微处理器在处理外设时一般是将 IO 空间与存储器空间统一编址（参见第 2 章 2.2.2 节），也就是说对 IO 设备控制寄存器和状态寄存器的访问都可以通过访存指令实现，因此通过调试器的访存功能同样可以配置和观察外设的功能。

嵌入式系统的调试相对于 PC 软件的调试是比较困难的，调试工具必须在远程主机上运行，通过某种通信方式与目标机连接，并通过在主机上运行交叉编译工具生成运行在目标系统上的目标文件。由于目标系统中常常没有进行输入/输出处理的必要的人机接口（这一点对于深嵌入的系统更是如此），因此需要在另外一台计算机上运行调试程序。这个运行调试程序的计算机通常是一台 PC，称为主机。在主机和目标机之间需要一定的信道进行通信。这样一个嵌入式系统的调试系统应该包括 3 部分，即主机、目标机、目标机和主机之间的通信信道。在主机上运行的调试程序用于接收用户的命令，把用户命令通过主机和目标机之间的通信通道发送到目标机，同时接收从目标机返回的数据并按照用户指定的格式进行显示。

在主机和目标机之间的通信信道通常是串行端口(PC 上的 COM 口)、并行端口（PC 上的 PRT 口，现在的 PC 已经很少支持这种并行口了）、以太网或者 USB。图 3-5 所示为嵌入式系统的调试架构。

嵌入式系统中常用的调试方法有以下几种：

（1）模拟器（Simulator）。

部分集成开发环境提供了模拟器，可方便用户在 PC 上完成一部分简单的调试工作，但是由于模拟器与真实的硬件环境相差很大，因此即使用户使用模拟器调试通过的程序也有可能无法在真实的硬件环境下运行，用户最终必须在硬件平台上完成整个应用的开发。Armulator 就是一种 ARM 公司 RVDS 集成开发环境中提供的指令集模拟器，下面将会详细介绍。

（2）驻留监控软件。

驻留监控软件（Resident Monitors）是一段运行在目标机上的程序，集成开发环境中的调试器软件通过以太网口、并行端口、串行端口等通信端口与驻留监控软件进行交互，由调试器发出命令通知驻留监控软件控制被调试程序的执行、读写存储器、读写寄存器、设置断点等。

（3）在线仿真器（In Circuits Emulation, ICE）。

由于在线仿真器中的处理器和被调试的目标处理器类似（相同的指令集），并且增加了监控处理器数据总线和地址总线的硬件逻辑，在线仿真器使用仿真头连接到目标系统的处理器插座（被仿真的目标处理器已经被拔掉了），完全取代目标机上的 CPU，可以仿真目标机上的微处理器芯片的行为，

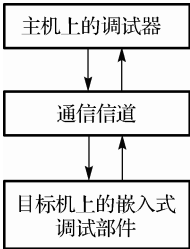


图 3-5 嵌入式系统的调试架构

提供更加深入的调试功能。传统的单片机开发通常采用在线仿真的方式进行调试，但对于引脚更多、速度更快的高性能 32 位嵌入式微处理器而言，这类仿真器为了能够全速仿真时钟速度高于 100MHz 的处理器，通常必须采用极其复杂的设计和工艺，因而价格比较昂贵，因此对于高性能处理器很少采用这种调试方式。

(4) 片上在线仿真器 (On Chip ICE 或者 Embedded ICE)。

为了解决传统在线仿真器难以调试高性能嵌入式微处理器的困难，现代处理器设计中往往将在线仿真器的相关硬件逻辑直接集成在处理器芯片中，这些硬件逻辑监测 CPU 的所有输入/输出信号，并比对调试器软件设置的断点地址或者观测点地址，如果相符则将 CPU 停下，以此实现断点和单步。片上在线仿真器通常通过 JTAG 接口与主机进行通信，接收调试器的命令，并传送目标处理器和存储器的内容给运行在主机上的调试器软件。

本节将以 ARM 公司的相关技术为例介绍嵌入式系统的各种调试方式，包括模拟器、纯软的驻留监控软件调试、传统的在线仿真调试、片上在线仿真调试等。

3.2.1 模拟器

基于模拟器的调试方法通常有两个功能：第一，用于设计的最初阶段，这期间设计人员会借助它来对初始代码进行评估。模拟器的指令都在 PC 上运行，不需要下载到目标系统上运行。开发人员需在设计进程的初期阶段，一般在获得硬件前的几个月，使用模拟器对复杂的系统进行建模，评估系统的性能瓶颈并指导硬件的架构设计。这属于软硬件协同设计的一部分。第二，对于基于操作系统之上的应用程序设计，首先由于这些应用程序往往基于操作系统之上，很少直接访问底层硬件；其次，由于应用设计人员众多，为每个程序员都配备相应的硬件开发板成本高昂；最后，硬件开发板的稳定性常常会造成应用程序开发的低效。这些都使得基于模拟器的应用开发是提高效率降低成本的好办法。模拟器的设计主要分为指令集模拟器和操作系统模拟器两种。

1. 指令集模拟器

基于指令集模拟器的调试方法主要是利用宿主机（通常为 PC）的资源来构建一个虚拟的目标机系统，该系统通过解释执行目标机的二进制代码来模拟执行目标系统的二进制程序。模拟器构建了一个虚拟环境，用户目标机的实时操作系统、应用程序等运行在虚拟环境中，这个虚拟环境包括了模拟出来的目标机 CPU、存储器、寄存器等，也就是模拟运行不同的微处理器核的指令系统。在实现上，有的指令集模拟器只是在功能上保证能够正确运行目标 CPU 的指令集，我们将其称为功能级的指令集模拟器；而有些模拟器不仅能够正确执行目标 CPU 的指令集，而且能够比较精确地模拟目标 CPU 的微架构和时序，我们将这类模拟器称为时钟精确型的指令集模拟器，后者通常用于 SoC 架构研究和 CPU 微架构研究。

以下是一些比较有代表性的指令集模拟器。

① ARMulator：ARM 公司自己推出的模拟器，模拟运行 ARM 内核的指令系统。

② Palm OS 模拟器：可以模拟运行 Motorola (FreeScale) 的 68K 微处理器的指令系统。

③ GEM5：是一款模块化的离散事件驱动的时钟精确型全系统模拟器，目前支持多种商用指令集 (ARM、ALPHA、MIPS、PowerPC、SPARC、X86)，同时支持在 ARM、ALPHA 及 X86 架构上运行 Linux 操作系统。GEM5 结合了密歇根大学的 M5 和威斯康星大学的 GEMS 两款模拟器中最优秀的部分，是一款高度可配置、集成多种 ISA (Instruction Set Architecture) 和多种 CPU 模型的体系结构模拟器。GEM5 是由企业界和学术界联合打造的一款体系结构级模拟器，包括 ARM、AMD、MIPS、HP、Google、Texas Austin、Wisconsin Masidon、MIT 等都参与贡献代码。

指令集模拟器的实现架构如图 3-6 所示，图中的 VM（虚拟机）即指令集模拟器，运行在 PC 的 Windows 操作系统上，虚拟机构建了目标系统硬件的一个虚拟环境，包括寄存器、存储器等，可以运行目标系统上微处理器的指令。针对目标系统所编写的程序，包括在目标系统上运行的嵌入式实时操作系统 RTOS、应用程序等可以在虚拟机上运行来模拟其在目标系统上运行的效果。

2. 操作系统模拟器

除了指令集模拟器之外还有操作系统模拟器，例如 Sybian OS 模拟器和国家 ASIC 中心自主研发的 ASIX OS 模拟器，它们在操作系统层次上进行模拟，将 Windows 的 API 封装成嵌入式操作系统的 API，如图 3-7 所示。在操作系统模拟器中，由于是在操作系统层面上进行仿真，应用程序运行的是 PC 的二进制代码（指令集模拟器中应用程序运行的是目标系统上微处理器的二进制代码），所以在目标系统上运行时需要重新编译。

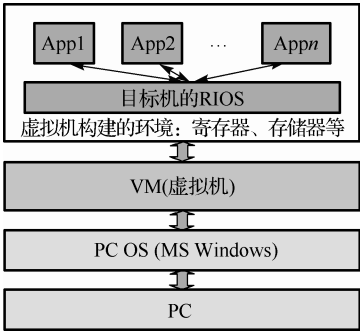


图 3-6 指令集模拟器的实现架构

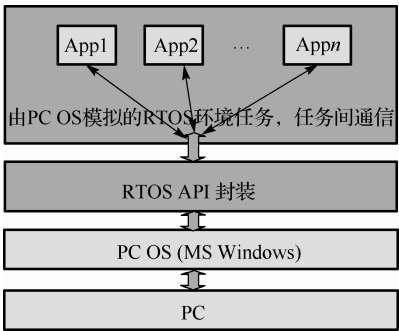


图 3-7 ASIX OS 模拟器

3. ARM 指令集模拟器 ARMulator

ARMulator 是 ARM 公司推出的一个指令集模拟器，运行在 PC 主机上，所以在没有硬件目标系统的情况下也可以通过 ARMulator 开发运行于特定 ARM 处理器上的应用程序。由于 ARMulator 可以报告各指令执行时的机器周期（虽然不是非常准确），所以也可以用来进行程序的性能分析。

ARMulator 被设计为允许容易地扩展软件模型，使之包括诸如 Cache、特殊存储器时序特性等系统特征。ARMulator 被很好地集成在 ARM 公司自己推出的 RVDS 集成开发环境套件中，可以无缝地与 AXD 调试器集成，这使得通过 AXD 调试器可以方便地为 ARMulator 加载程序并使用所有的调试方法（包括断点、观察点、单步等）。

ARMulator 是一套在主机系统上用软件模拟各种 ARM 微处理器核行为的程序，它可以在不同的精度级别上运行：

- 指令精度模型给出系统状态的确切行为，而不考虑处理器的精确时序特性。
- 周期精度模型逐个周期地给出处理器的确切行为，能比较准确地给出程序所需的时钟周期数。
- 时序精度模型给出信号在周期内的准确时序，能够考虑逻辑延时。

所有这些方式的运行都比实际硬件慢得多。但是，指令精度模型的速度相对来说最快，最适合于软件开发。简单地说，ARMulator 使得用 C 编译器或汇编器开发的 ARM 程序能够在没有 ARM 处理器连接的主机上进行测试和调试，使得程序执行所用的时钟周期数得以比较精确地测量，因此可实现对目标系统性能的评估。进一步而言，ARMulator 可以作为目标系统的完全的、具有大致时序精度的 C 模型来使用，它具有全部的 Cache 细节和存储器管理功能，可以运行某一操作系统。更进一步，ARMulator 还可以用作基于具有时序精度的 ARM 行为模型的内核，通过生成一个 VHDL 的“壳”作

为 ARMulator C 代码到 VHDL 环境的接口。总之, ARMulator 可以看作是一个 ARM 处理器的模拟器,不需要 ARM 处理器芯片就能调试和评价 ARM 代码。

随着系统芯片 (SoC) 设计技术的发展,芯片的复杂性日益增加,如何在设计之初准确地从整体上把握整个设计,对整个设计进行准确的评估,成为 SoC 设计中越来越突出的矛盾。ARMulator 在这方面提供了一个很好的解决方案,上文已经提到过它除了可以模拟出 ARM 公司各种内核的操作,是一种有效的源程序检验和测试工具之外,它还提供了一种扩展机制,可以在 ARMulator 的平台上用 C 语言描述硬件模型,通过其接口函数建立与 ARMulator 的联系,使得 ARMulator 中的指令集模拟器可以通过接口函数访问该模型。通过该硬件建模功能,用户可以在 ARMulator 存储器接口的基础上,开发基于 ARMulator 的 AMBA 总线模型和外设的接口模型。总线模型由 AHB、APB、总线仲裁机制和中断控制器 (INTC) 组成,外设的接口函数包括 AHB 的接口函数和 APB 的接口函数。利用接口函数,可以建立一个完整的、时钟周期精确的系统模型,包括 Cache、MMU、物理存储器、外围设备、操作系统和应用软件。由于这是系统的高层模型,因此最适合于完成设计方案的初始评估。使得架构设计人员能够在设计之初,将整个 SoC 系统比较准确地建立起来,这为系统的复杂应用和对模块设计进行评估优化工作提供了有效的方法,可以通过对各个模型的分析,确定各个模块的瓶颈所在,有针对性地进行优化,提高系统性能,使模块的性能最优化,为 RTL 级的设计确定整个芯片的架构。图 3-8 可以比较直观地反映在 ARMulator 基础上模拟器的改进,又称为虚拟原型。

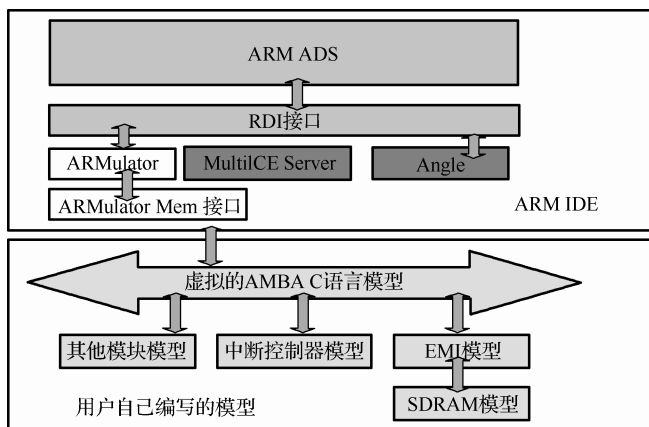


图 3-8 基于 ARMulator 模拟器的虚拟原型

在这个虚拟原型中,用户编写的虚拟 AMBA 总线可以接管所有的存储器访问,再通过总线接口函数与其他硬件模型进行通信,这样一来,ARMulator 就同用户自己编写的模型结合起来,实现了一个完整片上系统的模拟。所以用户通过修改或重写缺省的模型,几乎可以模拟所有基于 ARM 处理器的系统,用来调试与硬件相关的代码。当模拟外设时,用户要求 ARMulator 能够仿真外部事件的发生,为此,ARMulator 提供了两个例程辅助事件调度: ARM_Time 用来返回从系统复位到当前的时间片数目; ARMul_ScheduleEvent 允许在一定时间后调用某个程序。

下面通过一个仿真并行端口的实例来说明具体方法,要求该并行口产生一个中断,然后从一个文本文件中读出一个字符放入存储器中。实现步骤如下:

① 建立新的存储器模型。首先复制一份 ARMulator 的存储器模型,进行相应修改,创建 parallel.c 文件,添加两个变量:中断产生标志和指向要访问的文本文件的指针,添加一个新程序 parallel_set_irq 用来触发中断并设置中断产生标志为 1。

② 修改 models.h 和 makefile。在 models.h 中添加 MEMORY (Parallel) 作为新的存储器模型的根本

入口。编译 `parallel.c` 产生目标文件添加到 `makefile` 的目标文件列表中。

③ 添加新模型到 `ARMulator`，并在 PC 上使用微软的 `Visual C++` 重建 “`ARMULATE.DLL`”，产生新的 `ARMulator`。

④ 修改 `armul.cnf`，选择该存储器模型。

当应用程序访问一个特定存储器地址（如 `0x123450`）时，`ARMul_ScheduleEvent` 函数被激活。`ARMul_ScheduleEvent` 函数在 20000 个时间片后调用 `parallel_set_irq`，从而产生一个中断服务请求，系统进入中断服务程序（由应用程序设置中断向量安装 `ISR`），中断服务程序从另一个预定义的位置（如 `0x123460`）读出一个字符，并清除并行端口的中断源，返回应用程序。

这种方法与用户使用的 `ARM` 调试器无关，因为它是 `ARMulator` 的一部分，不属于调试器。这是一项非常有用的技术，可以扩展到用 `C` 语言为整个系统建立模型。例如，它可能模拟一个 `PDA` 系统（包括键盘、定时器、并行接口、液晶显示屏等）。

在设计比较稳定时，硬件开发就可以进入时序精确的 `EDA` 环境了，软件开发可以继续使用基于 `ARMulator` 的模型。

4*. GEM5 模拟器

在过去的数十年里，`M5` 和 `GEMS` 被数以百计的公开发表的论文所引用，并且相关平台源码被下载数万次。`M5` 和 `GEMS` 的高度集成、基于 `BSD`（`Berkeley Software Distribution License`）的授权方式以及 `GEM5` 基于社区的合作开发方式，使 `GEM5` 成为一个优秀的、被学术界及企业界广泛使用的全系统模拟器。`GEM5` 的总体目标是成为一款开源的致力于架构模拟的社区工具。实现这一目标主要有 3 个关键部分：灵活性、实用和可用性以及开发者的积极合作开发与互动。

（1）灵活性。

灵活性是任何优秀的模拟器必须具备的特点，比如架构师需要验证一个高层次概念来指导自己的特定设计，而这个高层的设计可能取决于不同层面的细节、模拟速度及精度等。例如，一个细粒度的门控时钟实验需要一个精细的 `CPU` 模型，但不需要多核模拟；同样，一个可扩展的互联模型需要多 `CPU` 模拟，但不需要支持 `CPU` 内部太多的细节。`GEM5` 的灵活性配置体现在提供多种 `CPU` 模型、系统模拟模式及存储模型。目前 `GEM5` 提供 4 种 `CPU` 模型：`AtomicSimple` 模型、`TimingSimple` 模型、`InOrder` 模型及 `O3` 模型。`AtomicSimple` 模型是一个简单的 `IPC`（`Instruction Per Cycle`）为 1 的模型；`TimingSimple` 模型和 `AtomicSimple` 模型类似但它具有存储子系统的时间模型；`InOrder` 模型是一个具有简单流水线的顺序模型；`O3` 模型是一个支持 7 级流水线、多发射和乱序执行的模型。

`GEM5` 提供两种模拟模式：`SE`（系统调用模拟）、`FS`（全系统模拟），每一个 `CPU` 模型都可以使用这两种模拟模式，`FS` 模式支持操作系统的运行和多种外设的模拟。`GEM5` 的存储子系统分为 `Classic`（经典）模型和 `Ruby` 模型，`Classic` 模型提供快速和灵活的存储子系统配置，而 `Ruby` 模型可以通过灵活配置来模拟不同的 `Cache` 一致性协议。最新的 `GEM5` 版本还将马里兰大学开发的 `DRAMsim2` `DDR` 存储器的模型集成进来，使得这个 `GEM5` 系统更加完备。

（2）实用和可用性。

`GEM5` 是基于 `BSD` 许可证管理的，是自由软件中使用最广泛的许可证之一。不同用户，包括学术研究人员、企业界工程人员及学生，可以在 `BSD` 许可证许可的范围内自由使用 `GEM5`。

（3）合作开发与互动。

`GEM5` 是一个社区开源项目，任何人都可以贡献自己的代码来修复缺陷或者添加新特性。

`GEM5` 的主要设计特点包括：面向对象、`Python` 集成、`DSL`（领域特定语言）、标准化接口。这些

设计特点都是很好的软件工程实践，它们对 GEM5 模拟器的设计非常有帮助。整个项目采用面向对象的方式进行设计，使用 C++ 和 Python 语言编写，这使 GEM5 具有高度的灵活性。GEM5 的所有组件被看作是一个 SimObject，它们共享一些行为，包括配置、初始化、统计和序列化。GEM5 内部的处理器核、Cache、互联结构、外设及像负载和进程上下文这类抽象的实体都是由 SimObject 来表示的。每一个 SimObject 都是由两个类来表示的，一个是继承自 SimObject 的 C++ 类，一个是 Python 类。Python 部分主要是给 SimObject 指定参数和执行一些脚本配置，而 C++ 部分封装了 SimObject 的状态和运行行为。对于不同的组件 GEM5 使用 Python 进行集成，Python 部分负责连接、初始化、配置和模拟控制组件，而 C++ 部分实现组件的具体行为。为了提供定制硬件的功能，GEM5 提供两种 DSL，一种用来定义 ISA，一种用来定义 Cache 一致性协议。标准化接口是面向对象设计的基础，GEM5 内部实现了端口接口和消息缓冲接口。端口接口是用来连接 GEM5 内部的两个对象，在 Classic 存储系统中，端口接口可以用来连接 CPU 和 Cache、Cache 和总线、总线与设备及内存，并且端口接口支持 Timing（时序）、Atomic（原子）、Functional（功能）模式以及用来确定拓扑及调试的接口。Timing 模式可以对访存时序进行详细建模，请求数据时给存储子系统发消息，而数据返回则是通过异步消息机制来实现的。Atomic 模式可以获取到一些计时信息，但它不是基于异步机制，而是以同步机制实现的，相比 Timing 模式，Atomic 模式速度快，但精度低。Functional 模式主要用来对模拟器状态进行更新，而不改变计时信息，主要用来进行调试、系统调用模拟和系统初始化时将数据加载到主存中。虽然 GEM5 的功能很强大，但是依旧需要开发者完善，如功耗模型、各种 ISA/CPU/存储模型的全面支持、并行化运行及检查点导入等。

5. GEM5 的 CPU 模型

AtomicSimple 模型是最简单的 CPU 模型，一个周期完成一条指令，存储模型比较理想化，访存操作为原子性操作，适用于快速功能模拟。其核心代码执行流程如图 3-9 所示，tick() 是每个 CPU 时钟周期都要执行一次，其首先会调用 setupFetchRequest 去构建一个对指令 Cache 的请求，同时会调用 translateAtomic 函数来进行 itlb 翻译；其次通过 sendAtomic 发出请求，同时从指令 Cache 或内存中取回指令；然后 preExecute 把指令翻译成微码；紧接着 execute 执行指令，对于访存指令，它会调用 readMem 和 writeMem 来读写数据，对于普通的计算指令，它会直接在 execute 函数内完成；最后执行 postExecute 来更新 CPU 内部一些计数器，如累加 numMemRefs、numLoad、numIntInsts 及 numFpInsts，以及更新跳转和函数返回次数等。

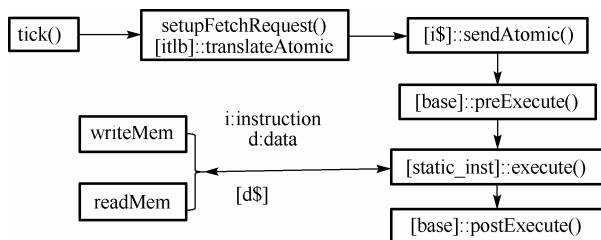


图 3-9 AtomicSimple 模型执行流程

TimingSimple 模型和 AtomicSimple 模型一样，也是无流水线的模拟，但是它使用了存储器访问时序模型，用于统计存储器访问延迟。其核心代码的执行流程如图 3-10 所示，advanceInst() 更新 PC 值，其次调用 fetch() 构造 request 请求，然后调用 translateTiming 进行 itlb 翻译，之后通过 sendFetch 和 sendTiming 把请求包发给指令 Cache；指令 Cache 的 slave 端口收到请求后执行 recvTiming，以回调函数的方式返回数据；数据成功返回，执行 completeIfetch()；获取指令失败，则返回 nack 给 CPU 端。获取到数据后，preExecute() 会把指令解释成微码，对于非访存指令调用 execute 执行命令，对于访存指令则执行类似获取指令的代码路径，指令执行结束后，postExecute() 会进行状态更新，最后回到 advanceInst() 执行下一条指令。

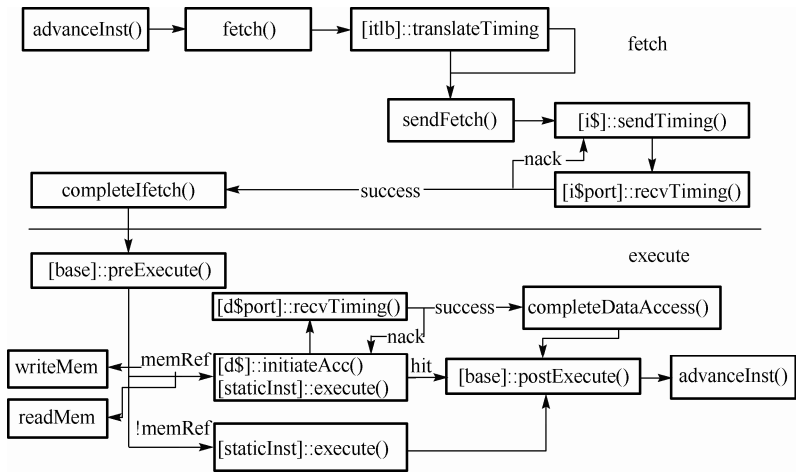


图 3-10 TimingSimple 模型执行流程

In-Order 模型是 GEM5 的新特性，其支持默认的 5 级流水：取指、译码、执行、访存、写回。它同时模拟了 Cache 部件、执行部件、分支预测部件等。

O3 模型支持 7 级流水，模拟了乱序执行和超量量执行的指令间依赖，以及运行于多 CPU 上的并发执行多线程。默认的 7 级流水分别是：取指、译码、重命名、发射、执行、写回、提交。它模拟了物理寄存器文件、指令队列、Load/Store 队列及 reorder buffer 等功能部件，同时可以对流水线管道间延迟、硬件线程数、IQ（Instruction Queue）/LSQ（Load-Store Queue）/ROB（reorder buffer）项数、FU（Function Unit）、物理寄存器重命名、分支预测算法、访存预取等参数进行配置。O3 CPU 模型支持的分支预测器包括局部预测器、全局预测器、tournament 预测器及分支目标缓冲区（BTB，Branch Targer Buffer）和返回地址栈。O3 模型的部分核心代码执行流程如图 3-11 所示，左图为 fetch 流程，右图为 Load/Store 流程。

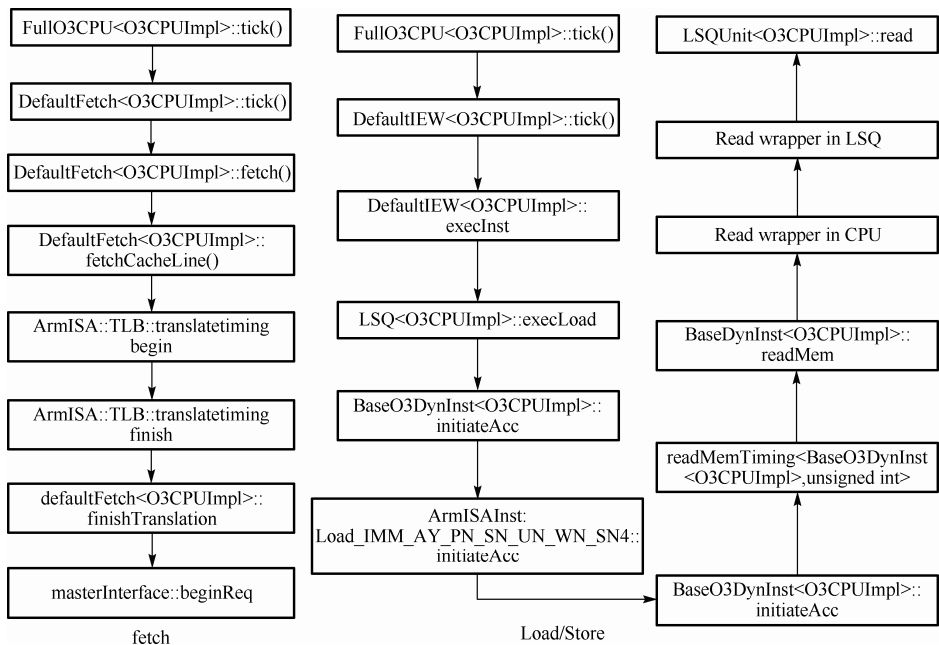


图 3-11 O3 模型部分核心代码执行流程

不同 CPU 模型、系统模式及存储模型的配置有不同的模拟精度，图 3-12 给出了不同配置下的速度与精度的平衡。其中 O3 CPU 模型，其相关微结构参数的配置与 ARM Cortex-A9 和 A15 非常接近。本章扩展阅读 [2]分析了 GEM5 在进行 CPU 架构模拟时的精度，其通过比对相同测试集在 Snowball 开发板（基于 Cortex-A9 双核）和 GEM5 ARM Detailed 模型，得出 GEM5 模拟器和真实开发板的数据误差在 1.39%~17.04%之间，这表明 GEM5 的仿真精度可以满足架构探索的需求。

处理器		存储子系统		
CPU 模型	系统模型	Classic	Ruby	
			Simple	Garnet
Atomic Simple	SE	速度		
	FS			
Timing Simple	SE			
	FS			
InOrder	SE			
	FS			
O3	SE			
	FS			精度

图 3-12 速度和精度配置上的权衡

图 3-13 给出了 GEM5 的工作流程，首先给 GEM5 指定基本的配置选项，然后可以在 Python 配置脚本中设定处理器内部的详细微结构参数，运行 GEM5，最后给出 config.ini 和 stats.txt 两个输出文件，config.ini 包含整个系统的配置描述，stats.txt 包含执行过程中微结构的执行信息，包括执行时间、CPI、Cache 缺失、分支跳转等。这两个输出文件也可以作为其他分析工具的输入来进行功耗等分析。关于 GEM5 的进一步内容，读者可以参阅本章扩展阅读[3]。

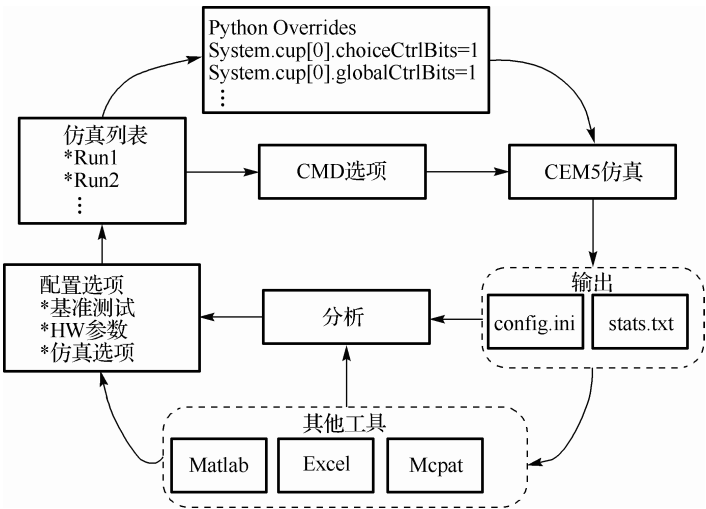


图 3-13 GEM5 的工作流程图

6. 模拟器的优缺点

指令集模拟器和操作系统模拟器都可以用于系统的前期算法分析和体系结构分析，例如 mp3 的播放测试、体系结构的模拟等。利用模拟器，可以模拟各个模块的功能，分析其瓶颈，进而实现各个功能模块的优化。利用模拟器还可以方便地进行应用程序的开发，由于上层应用程序是通

过驱动程序和硬件打交道，并不直接和硬件打交道，所以可以用模拟器来进行应用程序的开发，无需硬件的支持。

但基于模拟器的调试方法也有以下缺点：

- ① 模拟器难以真实地反映所有的外设，也就是说通过 PC 的外设模拟目标系统外设时会有难度。目标系统外设的中断在 PC 端模拟也是比较困难的。
- ② 模拟器难以进行实时系统仿真，因为程序是在模拟器上运行而不是在真实的目标系统硬件上运行。
- ③ 模拟器难以进行设备的驱动开发，由于模拟器难以真实地反映所有的外设，因此进行设备的驱动开发也是比较困难的。

3.2.2 驻留监控软件

1. 驻留监控软件调试方式简介

驻留监控软件调试方式的组成框图如图 3-14 所示，主要由两部分组成：一部分是主机（Host）端运行的调试器（Debugger），主要提供友好的用户界面，处理用户命令，与目标机通信；另一部分是目标机，目标机上被调试的程序放在 RAM 中，监控程序（Monitor）一般存放在 ROM 或其他非易失存储器中。监控程序提供软件调试功能（包括被调试程序的下载、断点、单步执行等功能），保存系统状态，读写 RAM，修改处理器状态（处理器寄存器）等基本功能。虽然不同的厂商推出的驻留监控软件的名字可能不同，有的称为天使程序（Angel），有的称为监控程序（Monitor），有的称为代理程序（Agent），还有的将这部分功能同 BootLoader 和 BSP 集成在一起（通常，BootLoader 和 BSP 中集成的调试功能要弱一些，比如只能装载程序，执行程序 and Dump 存储器，但不具备单步和断点功能），但是这些程序的基本原理基本一样。

驻留监控软件调试的基本原理是目标机上电复位后由监控程序接管系统并和调试主机建立通信链路，等待调试主机的命令请求。调试主机通过通信链路把被调试程序加载到目标系统的 RAM。监控程序通过修改目标系统处理器的 PC（程序计数器）来将系统控制权交给被调试程序。当控制权需要由监控程序交给被调试程序时，主机端调试器程序向目标机的监控程序发送修改 PC 命令，并给出 PC 的修改值。监控程序收到主机发过来的修改 PC 命令后，就按照主机的要求修改 PC，这样控制权就由监控程序交给被调试程序了。

监控程序用一条软陷指令（或者未定义指令）替代断点处原来的机器指令，当处理器读取这条指令时实际取到的是软陷指令，这样处理器就进入软件中断异常，由 ROM 中的监控程序接管这个异常，该监控程序把系统的一些状态发给主机，这样就完成了简单的断点功能。程序需要运行时再把软陷指令恢复成原来的指令，恢复处理器的 PC 和现场，让处理器恢复原来的运行，如图 3-15 所示。

主机端的调试器（Debugger）通过向监控程序发出读写存储器命令来完成对目标系统相关存储器的读写操作，监控程序按照命令的要求读取给定地址范围的存储器的内容，然后再通过通信接口把这些内容发回给主机端调试器，并由调试器调用显示函数把这些内容显示出来。怎样修改目标系统处理器的寄存器值呢？在进入监控程序后，该程序把处理器的各个寄存器值保存在固定地址的 RAM 中，

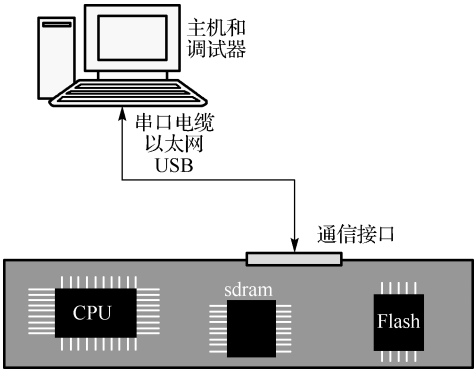


图 3-14 驻留监控软件调试方式

所以处理器的每个寄存器在 RAM 中都有一个 32 位的存储空间和它相对应。要修改处理器的寄存器时, 首先修改寄存器所对应的 RAM 内容, 然后再把 RAM 的内容加载到寄存器里。主机和监控程序间的硬件链路可用串口, 也可以用并口, 还可以用网口, 只是相应的驱动不同。同时为了保证通信的可靠性, 它们间也需要有保证通信可靠的协议。从上面的原理可以看出这种驻留监控软件调试方法只能对 RAM 程序调试 (因为无法把 ROM 中的指令替换成软陷指令), 同时需要占用目标系统的 ROM 空间的一部分 RAM 空间, 也需要占用目标系统的一个通信端口。这种调试方法的优点在于它不需要特别的硬件支持, 设计成本比较低。

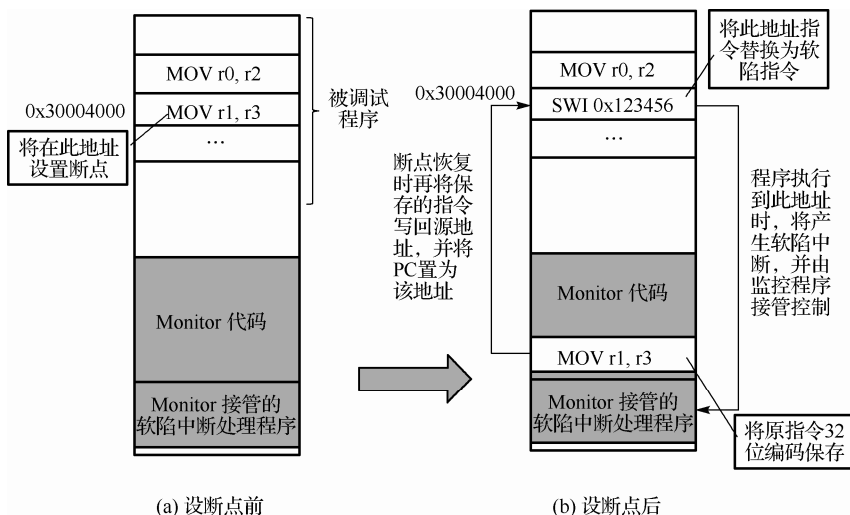


图 3-15 软件断点的实现

ARM 公司也曾在其 RVDS 开发套件中支持这种廉价的调试方式, ARM 将驻留在特定开发板上的监控程序称为 Angle。由于 Angle 支持 RDI 接口 (远程调试接口), 因此可以无缝地与 AXD 调试器连接, 并实现相应的调试功能。

2. 驻留监控软件调试方式的优缺点

驻留监控软件调试方式不需要额外的调试硬件, 成本比较低廉。目标系统上运行一个监控软件 (比如 ARM 系统所支持的 Angel), 这个监控软件占用一定的目标系统资源 (目标系统的内存等), 监控软件通过串口或者网口等与 PC 相连, 通过一定的通信协议 (ARM 系统支持的与 Angel 通信的协议是 ADP 协议) 与 PC 的调试器进行通信。这种驻留监控软件调试方式仅需要编写 PC 端的调试器软件和目标系统上的监控软件, 再通过串口或者网口连接 PC 和目标系统即可。可见, 驻留监控软件调试方式的实现非常简单, 是基于真实硬件的调试方式中最简单的一种。

驻留监控软件调试方式也有以下缺点。

① 监控软件需要占用目标系统的资源: 处理器、存储器和通信接口等。

② 无法在 ROM 区设置断点。因为驻留监控软件方式是通过把下一条要执行的指令替换成未定义指令来实现调试功能的, 所以只能修改 RAM 区的指令, 而不能修改 ROM 区的指令。

③ 无法设置数据观察点 (watchpoint)。由监控软件接管正常程序执行的唯一途径就是设置了指令断点把下一条要执行的指令替换成未定义指令, 当程序遇到未定义指令停止运行时才能够观察存储器的内容。这样就无法实现在指定的存储器处设置断点, 当存储器的内容发生变化时正常程序停止运行进入调试状态。

3.2.3 在线仿真调试

所谓在线仿真（In-Circuit Emulation, ICE）是指目标系统中的处理器被仿真器中的处理器取代，被调试的程序实际上由与目标系统处理器同指令集的仿真器中的处理器执行。在线仿真器本身就是一个嵌入式系统，有它自己的处理器、RAM、ROM 和自己的嵌入式软件。仿真器上的处理器可以是一个与目标板处理器相同的芯片，也可以是一个有更多引脚的变型芯片（对内部状态有更高的可观察性，但应该与被仿真的目标处理器具有相同的指令集）。仿真器上还有缓冲器，以便将处理器地址总线和数据总线上的活动复制到跟踪缓冲器（保存若干周期之内所有引脚上每个时钟周期的信号）中。在线仿真器通过硬件的方法侦测处理器上的所有信号，包括断点的设置等也是通过逻辑控制硬件来完成的。

利用在线仿真器可以完成比驻留监控软件调试更为强大的仿真功能。尽管利用驻留监控软件调试方式可以在程序中设置断点，但是这些断点只能到指令提取级别，也就是相当于“在提取指令前停止运行”。相比之下，在线仿真器支持硬件断点。硬件断点允许响应多种事件来停止运行程序。这些事件不仅包括指令提取，还有内存和 I/O 读写以及中断。例如，利用软件断点只能调试 RAM 中的程序（因为要把指令改为软件断点指令或软陷指令），而利用硬件断点还可以调试 ROM 中的程序（无需更改指令，只需侦测处理器发出的地址即可）。除此之外，这种调试方法不占用目标资源，所以在单片机系统中得到了广泛的应用。

但是这种传统的在线仿真器有一些明显的缺点：

① 现代嵌入式微处理器的 CPU 内核往往深嵌入在芯片内部，CPU 的地址总线和数据总线可能根本不引出到芯片外部，因此为了能够监测这些信号，仿真器内部的处理器必须将这些信号引出，这一方面使得仿真器内部的处理器和被调试的目标处理器实际上是不同的处理器；另一方面，单独为仿真器设计的处理器由于用量相对而言要少得多，使得其单颗成本非常高昂。

② 在线仿真器拥有自己的目标处理器、RAM、ROM 和自己的嵌入式软件，对于面向 32 位高性能嵌入式微处理器的高速在线仿真器而言，这些都是非常昂贵的，增加了调试成本。

既然通过片外的控制逻辑可以实现对芯片的调试，那为什么不把这个控制逻辑做在芯片内呢？每颗处理器都带有自己的调试逻辑，虽然这会增加一些处理器的芯片面积从而增加芯片成本，但是对于 32 微处理器而言，这点增加的成本在整个处理器成本中占的比重并不大。基于这样的理念，人们提出了片上在线仿真技术，下一小节将详细介绍这种调试方法。。

3.2.4 片上在线仿真调试

片上在线仿真（On Chip In Circuits Emulation 或 Embedded ICE）的调试方法是目前 32 位嵌入式微处理器比较常用的调试方法，其调试原理是通过在嵌入式微处理器芯片内部集成一个嵌入式（片上）在线仿真模块以及相关接口电路（比如 JTAG 接口电路）来实现在线仿真器的功能。这样做的好处是显而易见的：

① 由于现代嵌入式微处理器为了便于 PCB 生产测试和芯片制造测试通常都已经集成了 JTAG 接口，通过标准的 JTAG 接口来访问芯片上集成的 ICE 模块，因此不需要增加额外的接口和宝贵的芯片管脚。因为增加管脚往往意味着芯片封装尺寸的增加，而这将进一步增加芯片的成本。

② 这种芯片上的在线仿真器所监测和调试的处理器就是用户实际使用的处理器，可以发现一些实际的问题。本质上来说这已经不是仿真，而就是“真”的。

③ 芯片上在线仿真由于和处理器集成在同一颗硅片上，因此可以与被调试 CPU 以同样的速度运行，可以实现真正的 On Speed 调试，而且也不需要像外接仿真器那样处理复杂的高速电路设计问题，因此其成本比传统的高速在线仿真器要低许多。

④ 在进行调试时不需要在目标系统上运行程序，因此对于一个“裸的”目标系统也可以进行调试。而驻留监控软件调试方式则需要在目标系统上运行监控程序，这就需要一个可以工作的最小系统。

⑤ 除了可以在 RAM 中设置断点外，还可以在 ROM 中设置断点。

⑥ 由于采用 JTAG 接口进行通信，不需要通过占用特定的通信接口与目标板系统进行通信。

ARM 公司的 Multi-ICE 以及 EmbeddedICE 就是基于 JTAG 的调试系统。这个系统中 Multi-ICE 是用于连接 PC 主机与目标系统的协议转换器，其负责进行 PC 并口或 USB 数据与 JTAG 信号的转换。来自 PC 上运行的调试器命令经过协议转换后，以 JTAG 扫描链的方式输入到集成在处理器上的 Embedded ICE 模块。而 Embedded ICE 所读取的处理器信息也将通过 JTAG 扫描链输出到片外，经协议转化后传输给 PC 端的调试器软件。这类调试系统利用 ARM 处理器中的 JTAG 接口以及一个嵌入的调试单元和主机上的调试器进行通信，完成下面的工作：

- 实时地设置基于指令地址值或者基于数据值的断点。
- 控制程序单步执行。
- 访问且可以控制 ARM 处理器内核。
- 访问 ASIC 系统
- 访问系统中的存储器。
- 访问 I/O 系统。

在调试目标系统时，首先要通过一定的方式使目标系统进入调试状态。在调试状态下就可以完成各种调试功能，例如查看处理器状态、查看和修改存储器内容等。ARM7TDMI 可以通过下面的方式进入调试状态：

- 通过设置程序断点（breakpoint）。
- 通过设置数据断点（watchpoint）。
- 相应外部请求进入调试状态。

在目标程序中特定的位置设置断点后，当该位置处的指令进入指令流水线时，ARM 内核将该指令表示为断点指令。当程序执行到断点指令时，处理器进入调试状态；当断点设置在条件指令上时，不管该指令执行的条件能否得到满足，当该指令到达执行周期时，处理器也都会进入调试状态。注意，此时断点指令没有被执行。

在某条指令上设置了断点后，如果在该指令到达执行周期之前程序发生了跳转，或者发生了异常，断点指令就可能得不到执行。这种情况下，处理器将会刷新指令流水线，从而使得处理器不会在该断点指令处进入调试状态。

当用户设置了数据断点时，目标系统中的调试部件将会监视数据总线。如果用户设置的条件得到了满足，处理器将会在执行完当前指令后进入调试状态。如果当前指令是 LDM 或 STM，处理器将会在完成所有指令操作后进入调试状态。

3.2.5 跟踪（Trace）技术

据统计，在一款 SoC 的生命周期中，主要的工程成本来自软件。因此如果希望产品取得成功，开发者必须在软件方面下足功夫，这意味着开发更高质量的软件代码，尽可能地修正程序中的错误，并尽可能地优化软件性能。工欲善其事，必先利其器！为了达到上述目标，开发人员需要更加强大的调试工具。这些调试工具不仅可以帮助开发人员修正程序中的错误，还应该可以为他们进行系统性能分析。我们前面介绍的片上在线仿真调试技术是典型的 run-control 调试方式，其过程可概括为：设置断点→程序暂停→观察程序运行状态→返回继续运行。但是这种调试方式具有非常大的侵入性，它改变

了程序和系统的执行状态，比如可能引起 Cache 的全部刷新。因此，这种调试模式至少在以下三种情况下会变得难以适应开发者的需要：第一，调试通信类的协议软件时。我们可以在接收到一个网络传输包时停下处理器来观察这个包，但在停下系统的同时，网络上传输的数据并没有停止，这就会使我们在断点发生的时候错过网络上传输的信息。第二，调试多核系统的软件时。现代的 SoC 越来越复杂，往往集成了多个处理单元，这些处理单元可以是同构的，比如 CMP；也可能是异构的，比如 CPU + DSP + GPU + VPU。不管是何种情况，当我们停下一个 CPU 时，其他的处理器并没有停下，这使得对整个系统的观察和控制变得异常困难。更何况，我们似乎没有办法为 GPU 设置断点。第三，当需要知道总线的负载情况以及其他硬件信息时。基于复杂 SoC 的软件优化已经远远不只是针对 CPU 上的软件优化，开发人员还需要知道总线的负载情况、存储系统的性能等硬件信息，而这些信息显然是无法通过片上在线仿真获得的。

跟踪 (Trace) 技术可以解决上述的问题。它的主要思想是：在调试时用户首先设置监视点，监视点一旦被触发，内核的当前状态信息（如 PC、Data、Address 等）经过处理后被写回特定缓存，调试器可以读取缓存中的信息，然后解析这些信息从而得知监视点触发后内核的运行状态，进而达到了了解运行过程的目的。与断点调试技术最大的不同之处在于：第一，在跟踪的过程中，并不需要停下当前正在运行的处理器以及其上的软件。第二，跟踪的过程通常需要记录一段时间内的动态执行过程。我们可以用一个形象的例子来说明两者的不同，断点调试技术就像是给系统拍摄的一张照片，它只反映了当前的系统状态。而跟踪技术更像是给系统拍摄的一段视频，它反映了系统在一段时间内的动态特性。

我们可以通过软件的方式实现跟踪。最简单且最常用的跟踪方式其实就是 `printf` 函数。程序员可以在代码中插入 `printf` 或者 `printk` 函数将当前的状态打印出来，这样程序员就可以利用打印信息跟踪程序动态运行的过程。在 Linux 系统中，程序员还可以利用系统已经实现的精灵 (daemon) 程序，比如 `syslogd` 和 `klogd` 来记录系统和内核的消息。用户还可以通过 `strace` 命令显示一个用户空间的程序所调用的所有系统调用，以及这些调用的参数和返回值。而 `ltrace` 作为一款调试应用可以检测一个用户程序所调用的库函数，以及该应用所接收的所有信号和系统调用。Linux 内核还提供了可以让用户侵入探测的机制，比如 `Markers` 和 `Tracepoints`。这些机制采用钩子的方法，允许用户探测系统行为^①。

虽然采用软件的跟踪机制已经比较成熟而且功能强大，但软件跟踪方法依然有不可逾越的困难。首先是性能的开销。软件跟踪往往需要增加额外的执行时间，比如 `printf` 函数在一个大循环中的开销还是相当可观的。其次，软件的跟踪方法本质是侵入的跟踪技术，插入的跟踪代码可能会改变程序原来的运行状态。比如，由于跟踪代码的加入，原来能很好适应 Cache 结构软件可能变得不是非常匹配了。再次，高延迟。比如，由于 `printf` 和 `printk` 函数的延迟比较大，它们不适合在临界区代码和中断处理函数中被调用（且不说在有些编译器提供的库函数中，`printf` 函数本身是不可安全重入的^②）。最后，采用软件方法所获得的跟踪信息相对比较有限。现代 SoC 的复杂性使得图形处理主要由 GPU 完成，媒体数据处理主要由多媒体引擎完成，而信号处理则可能依赖集成的 DSP 完成。所有这些处理单元又是通过高速互联网络进行连接的。为了更好更深入地知道系统中发生的事件，仅仅通过软件监测的 CPU 行为就显得不够了。

为了适应对整个 SoC 进行跟踪的需求，芯片设计者必须在芯片内部增加相应的跟踪模块。通过硬件的跟踪模块监测和跟踪不同硬件模块的运行状态。我们将在下一小节以 ARM 公司推出的 CoreSight 框架为例，比较详细地介绍基于硬件的跟踪技术。

① 对于 Linux 软件跟踪技术感兴趣的读者可以参考 www.lttng.org 网站。

② 关于函数的重入问题可以参见本书的 7.2.5 节。

(2) 跟踪宏单元 (Trace Macrocells): 用于在 SoC 中提供全面的非干预性可见性。为什么需要嵌入式跟踪宏单元呢? 在进行调试或优化嵌入式系统时, 嵌入式开发人员有两个主要选择: 常规调试 (也就是断点调试) 和实时跟踪。在常规调试中, 通常需要设置断点和/或观察点以停止处理单元, 从该处使用调试连接来检查或修改寄存器或内存, 并执行一个单步以了解程序的工作方式。尽管开发人员可通过常规调试来控制执行并调试代码, 但常规调试存在一些缺点: 干预性, 即调试将控制系统行为; 需要停止处理器。但对于某些应用, 可能无法停止处理器 (例如硬盘、汽车等); 非实时, 即无法用于实时调试运行的软件; 性能不可见, 即无法观察软件性能。在实时跟踪情况下, 系统运行时跟踪宏单元收集指令和/或数据传输, 压缩这些信息, 并可以实时地将这些信息存储在芯片内的缓存或传输到芯片外, 以进行后续处理。通过使用 CoreSight 嵌入式跟踪宏单元, 开发人员可以实时分析软件如何在平台上运行, 以加快产品开发速度并提供高质量的优化软件。ARM 公司根据用户的不同需要, 可以提供不同的跟踪宏单元, 芯片设计人员可以在 CoreSight 架构下集成不同的跟踪宏单元。常见的跟踪宏单元包括:

- AHB 跟踪宏单元 (HTM, AHB Trace Macrocell): 用于获取 AHB 总线跟踪信息, 包括总线的层次、存储结构、时序、数据流和控制流等。HTM 提供有关 AHB 总线的地址和数据跟踪信息。可以将 HTM 中的信息与调试器配合使用, 以便轻松准确地调试基于 AHB 的嵌入式系统。HTM 提供了广泛的资源以使事件识别功能生成触发事件。HTM 通过 AMBA 跟踪总线 (ATB) 生成输出的跟踪数据。跟踪调试功能是非干预性 (非侵入) 的, 可以使用 APB 接口来控制 HTM。
- 嵌入式跟踪宏单元 (ETM, Embedded Trace Macrocell): ETM 宏单元为 ARM 微处理器提供实时指令跟踪和数据跟踪。跟踪软件工具使用 ETM 生成的信息重建全部或部分程序的执行情况。嵌入式跟踪单元 (ETM) 是一个实时跟踪模块, 它提供了处理器指令和数据跟踪。ETM 是 ARM CoreSight 调试和实时跟踪解决方案的不可或缺的一部分。
- 程序跟踪宏单元 (PTM, Program Flow Trace Macrocell): PTM 宏单元是一个实时跟踪模块, 它提供了处理器指令跟踪。PTM 是 ARM CoreSight 调试和实时跟踪解决方案的不可或缺的一部分。
- 测量跟踪宏单元 (ITM, Instrumentation Trace Macrocell): 是一个由软件驱动的跟踪单元, 其输出的跟踪信息可以由软件设置, 包括 Printf 类型的调试信息、操作系统以及应用程序的事件信息等。此外, 该宏单元还提供粗略的时间戳功能。

(3) 多处理器跟踪和跟踪链接部件: 用于事件跟踪数据的连接、触发和传输, 典型的有:

- 同步 ATB 桥: 同步 ATB 桥提供一个寄存器片, 可通过添加管道阶段来实现时序收敛。具有两个 ATB 接口, 用于传递跟踪源发出的控制信号。
- 复制器 (Replicator): 复制器可以将两个跟踪接收器连接在一起, 并在相同的传入跟踪流上运行。输入跟踪流是两个 ATB 端口上的输出, 这两个端口可以单独运行。
- 跟踪聚合器 (Trace Funnel): 跟踪聚合器将最多 8 个跟踪源合并到一个聚合器中。静态仲裁方案选择要在任何瞬间通过的输入跟踪流。静态仲裁允许在两次跟踪会话之间重新组织辅助端口优先级。聚合器可以链接在一起, 而将一个聚合器的 ATB 输出连接到另一个聚合器的 ATB 输入端口。这样, 就可以提高输入数量以及连接独立的系统。

(4) 收集跟踪和跟踪接收器: 是芯片上跟踪数据的终点, 典型的有:

- 跟踪端口接口单元 (TPIU, Trace Port Interface Unit): TPIU 是一个 ATB 辅助接口, 可从芯片中提取跟踪数据。它用作片上跟踪数据以及跟踪端口分析器 (TPA) 捕获的数据流之间的桥。TPIU 中的格式设置程序将源数据和 ID 合并为单一数据流以允许序列化数据, 从而在检测到触发器时插入触发数据包, 然后通过 Trace Port 传送到片外。
- 嵌入式跟踪缓冲区 (ETB, Embedded Trace Buffer): 一个 32 位的 RAM, 作为片内跟踪信息缓冲区。

- 串行线输出 (SWO, Serial Wire Output): SWO 是一个类似于 TPIU 的跟踪接收器, 它只能跟踪一个来源, 即 ITM。它通过单针接口从芯片输出数据流。
- HSSTP 体系结构规格指定了串行传输端口 (STP) 以替代现有的并行数据输出口, 它适用于将高带宽数据传输到芯片外。HSSTP 规格减少了 ASIC 针数, 增加了可用的带宽, 在某些情况下还会降低硅面积。

2. CoreSight 调试访问端口

CoreSight 调试访问端口 (DAP, Debug Access port) 是桥接外部调试器和 SoC 内多个 IP 核的一个组件。它将外部的 JTAG 数据传输转化成各种不同的内部传输信号。DAP 接收来自 JTAG 的命令, 再通过 DAP 内部的标准总线对 JTAG_AP、AHB_AP 以及 APB_AP 进行控制和访问。JTAG_AP 接收来自标准总线的传输, 然后翻译成 JTAG 指令通过这些指令控制 IP 核的 TAP 控制器。AHB_AP 是一个 AHB 总线的主设备, 通过它可以访问到任何接到 AHB 总线上的从设备, 比如共享存储器。通过 APB_AP 只能访问 Debug APB 总线, 当然也可以通过系统 AHB 总线访问 Debug APB 总线, APB_MUX 从这两条访问路径中选出一条送到 Debug APB 总线上, 通过 Debug APB 总线可以控制和访问到所有支持 APB 总线的组件。

许多 IP 核都支持 JTAG 调试接口, 因此通过 JTAG_AP, CoreSight 不仅兼容 JTAG 访问, 而且 JTAG_AP 提供了 8 个 JTAG 接口, 实现 JTAG 多核调试。几乎所有的控制寄存器都可以通过 AHB 总线来访问, 通过 AHB_AP 可以轻易配置或读取这些寄存器的信息。现在越来越多的 IP 提供了 APB 调试接口, 这也极大地简化了调试配置过程。

基于 CoreSight 的调试设计有如下很多优势:

- ① 即使在处理器运行时, 也可以查看存储器和外设的寄存器内容。
- ② 使用单一调试器, 就可以控制多核系统的调试接口。例如, 如果使用 JTAG, 则只需要一个 TAP 控制器, 不管芯片中有多少个 IP 核都一样。
- ③ 内部的调试接口是以总线方式设计的, 因此非常有弹性, 也使得为芯片的其他部分设计附加的测试逻辑变得容易。

在标准的 CoreSight 系统中, 会给调试总线单独开辟一个地址空间。图 3-17 所示是 CoreSight 系统设计概念图。

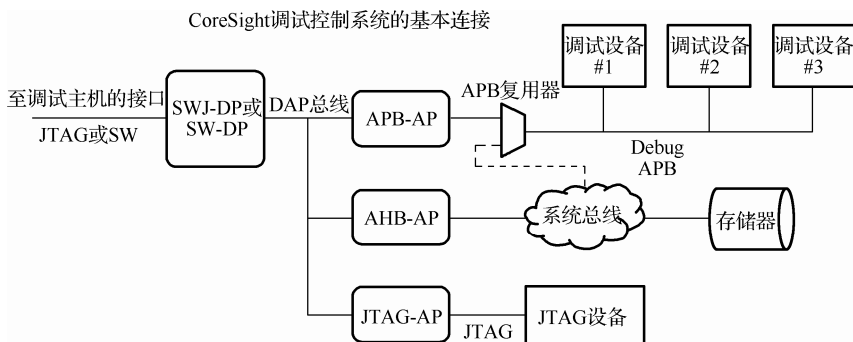


图 3-17 CoreSight 系统设计概念图

外部调试主机通过 JTAG 接口连接到 CoreSight 的调试接口, 调试接口将 JTAG 指令翻译成 DAP 总线上的信号。三个 AP 接口都可以从 DAP 总线上获取调试命令。Debug APB 是 APB 的近亲, 芯片设计人员可以很容易地在它上面挂接多个调试宏单元, 从而使得调试系统可大可小, 伸缩性很强。通

过 APB_AP 可以访问到 Debug APB 总线上的所有调试宏单元，而调试设备#1、调试设备#2、调试设备#3。而 ARM 处理器内核、DSP 数字信号处理器以及片上存储器都是挂载在 AHB 总线上的，因此通过 AHB_AP 可以访问到这些设备。同时 AHB_AP 也可以访问到 Debug APB 总线上的设备。此外，只要有 JTAG 接口的设备（IP 核）都可以通过 JTAG_AP 访问。通过 DAP 上提供的 APB 访问端口、AHB 访问端口和 JTAG 访问端口，运行于 PC 端的 CoreSight 调试器可以访问到整个 SoC 芯片上的所有资源。

图 3-18 就是 AHB_AP 访问 Debug APB 总线上的调试设备的一个例子。

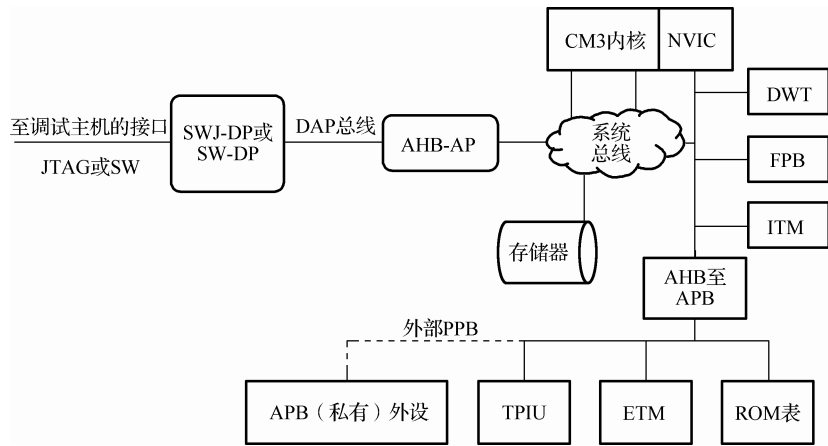


图 3-18 AHB_AP 访问 Debug APB 总线上的调试设备

挂载在 AHB 总线的设备有 ARM CortexM3 内核。ARM CortexM3 实际的调试功能由 NVIC 和若干调试组件来协作完成。调试组件包括 FPB、DWT、ITM 等。NVIC 中有一些寄存器，用于控制内核的调试动作，如停机、单步；其他的一些功能块则控制观察点、断点，以及调试消息的输出。这些调试组件都可以通过 AHB_AP 来进行配置。

而在这个调试系统中，ETM、TPIU、ROM 表等这些 CoreSight 的调试组件以及一些 APB 私有外设和 AHB 上的设备是使用同一套存储器映射的。在这个调试系统中的一个特点就是共用 AHB_AP 接口来访问 AHB 和 Debug APB 总线上的设备，因此需要一个特别的转换接口，那就是 AHB2APB 桥。AHB2APB 桥是 AHB 总线上的从设备，同时又是 Debug APB 总线上唯一的主设备。

3. CoreSight 中的实时跟踪

在 SoC 中实现内核的实时跟踪涉及两个关键技术：信息的采集和信息的传输。

跟踪电路的面积（包括控制逻辑和缓存）开销应尽量减少。在对处理器内核进行跟踪时不可能且没有必要将所有指令和数据都保存下来。调试人员通常并不需要查看处理器的所有指令和数据，而只关心特定信息。因此，跟踪数据采集功能应当能够根据调试需要有选择地采集信息。在 ARM CoreSight 架构中使用 ETM 模块实现对内核的数据采集。通过可配置的触发（trigger）和过滤（filter）条件，ETM 能够灵活地控制在何种情况下对哪些数据进行采集和压缩。

采集后的数据必须通过一定的通信机制传输到芯片外的数据分析终端。因此，在 ARM CoreSight 架构中定义了跟踪总线（ARM Trace Bus，ATB）、跟踪聚合器（Trace Funnel）、ATB 复用器（ATB Replicator）、跟踪接口单元（Trace Port Interface Unit）及跟踪缓存（Embedded Trace Buffer）。这些模块结合起来，为 ETM 产生的跟踪数据提供传输的通道，如图 3-19 所示。

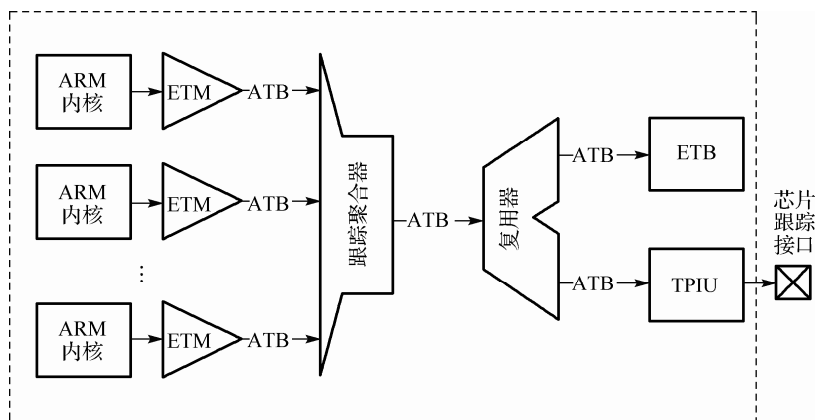


图 3-19 ARM CoreSight 架构中内核跟踪数据传输

由图 3-19 可以看出，在多个内核的情况下，跟踪聚合器先将来自各个 ETM 的 ATB 信号聚合成一个 ATB 接口；然后通过一个 ATB 复用器连接到 TPIU 或者嵌入式跟踪缓存（这个缓存为可选）；最后，数据通过 TPIU 发送到芯片的跟踪接口（供片外的跟踪终端进行采集）。在这一架构中，ETM 为跟踪数据源，TPIU（或 ETB 为数据终点）。ATB 复用器只是简单地将 ATB 信号分成两路，它可以被看作数据终点的一部分。^①

4. CortexM3 内核调试

CortexM3 在内核水平上搭载了若干种调试相关的特性，最主要的就是程序执行控制，包括停机（halting）、单步执行（stepping）、指令断点、数据观察点、寄存器和存储器访问、性能速写（profiling）以及各种跟踪机制。

CortexM3 的调试系统基于 ARM 最新的 CoreSight 架构。不同于以往的 ARM 处理器，内核本身不再含有 JTAG 接口。取而代之的是 CPU 提供称为“调试访问接口”（DAP）的总线接口。通过这个总线接口，可以访问芯片的寄存器，也可以访问系统存储器，甚至是在内核运行的时候访问。对此总线接口的使用，是由一个调试端口（DP）设备完成的。DP 不属于 CortexM3 内核，但它们是在芯片内部实现的。目前可用的 DP 包括 SWJ-DP（既支持传统的 JTAG 调试，也支持新的串行线调试协议）和 SW-DP（去掉了对 JTAG 的支持）。另外，也可以使用 ARM CoreSight 产品家族的 JTAG-DP 模块。芯片制造商可以从中选择一个，以提供具体的调试接口（通常都是选 SWJ-DP）。

通过集成在片内的“嵌入式跟踪单元”（ETM）实时收集处理器内核的执行信息（也就是 Trace），将这些信息通过“跟踪端口接口单元”（TPIU）输出到内核的外部，并最终输出到调试主机。

在 CortexM3 中，调试动作能由一系列的事件触发，包括断点、数据观察点、fault 条件，或者是外部调试请求输入的信号。当调试事件发生时，CortexM3 可能会停机，也可能进入调试监视器异常处理程序。具体如何反应，则根据与调试相关的寄存器的配置而定。

所有这些调试组件都可以由 DAP 总线接口来控制，CortexM3 内核提供 DAP 接口。此外，运行中的程序也能控制调试组件。所有的跟踪信息都能通过 TPIU 访问。

^① 在最新版本的 CoreSight 中，跟踪信息的汇总输出统一由一个被称为跟踪内存控制器（Trace Memory Controller, TMC）的新的模块集中处理。

3.3 基于 JTAG 接口的片上在线仿真

3.3.1 JTAG 简介

在 20 世纪 80 年代，联合测试行动组（Joint Test Action Group，JTAG）起草了边界扫描测试（Boundary-Scan Testing，BST）规范，后来在 1990 年被批准为 IEEE 标准 1149.1-1990 规定，简称 JTAG 标准。JTAG 最早被开发出来的目的是为了了解决 PCB 电路板在完成焊接后如何测试所有被焊接的芯片管脚的连通性问题；后来 JTAG 也被用于集成电路芯片（尤其是数字电路芯片）在完成封装后的 Final Testing 的测试矢量加载和测试结果的输出；JTAG 的第三个重要作用是作为访问片上在线仿真（Embedded In Circuits Emulation，Embedded ICE）逻辑的通信接口。我们在本节重点介绍 JTAG 作为调试接口的功能。这种通过一个接口实现不同功能的做法，最大程度地复用了芯片管脚，降低了芯片的封装成本。

1. 边界扫描

在 JTAG 调试中，边界扫描（Boundary-Scan）是一个很重要的概念。边界扫描技术的基本思想是在靠近芯片输入/输出管脚上增加一个移位寄存器单元。因为这些移位寄存器单元都分布在芯片的边界上（周围），所以称为边界扫描寄存器（Boundary-Scan Register Cell）。当芯片处于调试状态时，这些边界扫描寄存器可以将芯片和外围的输入/输出隔离开来。通过这些边界扫描寄存器单元，可以实现对芯片输入/输出信号的观察和控制（这也是开发 JTAG 标准实现芯片管脚焊接连通性测试的初衷）。对于芯片的输入管脚，可以通过与之相连的边界扫描寄存器单元把信号（数据）加载到该管脚中去；对于芯片的输出管脚，也可以通过与之相连的边界扫描寄存器“捕获”（Capture）该管脚上的输出信号。在正常的运行状态下，这些边界扫描寄存器对芯片来说是透明的，正常的运行不会受到任何影响。芯片输入/输出管脚上的边界扫描寄存器单元可以相互连接起来，在芯片的周围形成一个边界扫描链（Boundary-Scan Chain）。一般的芯片都会提供几条独立的边界扫描链，用来实现完整的测试功能。边界扫描链可以串行地输入和输出，通过相应的时钟信号和控制信号，就可以方便地观察和控制处在调试状态下的芯片。边界扫描结构如图 3-20 所示。

早期的扫描链主要是用于测试芯片焊接到 PCB 板上的连通性，因此主要集中于芯片的输入/输出管脚（Pin），这也是这些扫描链被称为边界扫描链的原因。为了测试芯片功能的正确性（主要用于芯片制造的量产测试），除了边界扫描链外，还需要在芯片内部功能模块（比如 CPU）的输入/输出信号（注意，不是芯片管脚，而是这个模块的信号线）外围构建扫描链，以实现对这些信号的观测与控制。因此现代数字集成电路中往往集成了多条不同的扫描链，下文中所说的边界扫描链实际包含了上述两种类型（也就是芯片的边界和内部功能模块的“边界”）。

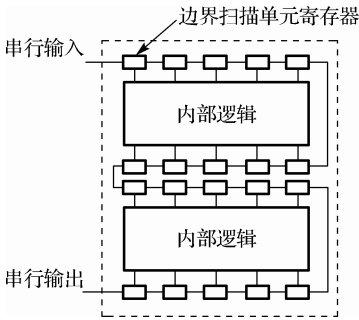


图 3-20 边界扫描结构

2. TAP（TEST ACCESS PORT）

利用边界扫描链可以实现对芯片的输入/输出进行观察和控制，而对于这些边界扫描链的管理和使用则需要用到测试访问端口 TAP（Test Access Port）来完成。

在 IEEE 1149.1 标准中，寄存器被分为两大类：数据寄存器（DR-Data Register）和指令寄存器

(IR-Instruction Register)。边界扫描链属于数据寄存器中很重要的一种。边界扫描链用来实现对芯片的输入/输出的观察和控制。而指令寄存器用来实现对数据寄存器的控制。例如，在芯片提供的所有边界扫描链中，选择一条指定的边界扫描链作为当前的目标扫描链，并作为访问对象。

TAP 是一个通用的端口，通过 TAP 可以访问芯片提供的所有数据寄存器(DR)和指令寄存器(IR)。对整个 TAP 的控制是通过 TAP Controller 来完成的。TAP 总共包括 5 个信号接口 TCK、TMS、TDI、TDO 和 TRST。

① TCK (Test Clock Input): TCK 为 TAP 的操作提供了一个独立的、基本的时钟信号，TAP 的所有操作都是通过这个时钟信号来驱动的。TCK 在 IEEE 1149.1 标准里是强制要求的。

② TMS (Test Mode Selection Input): TMS 信号用来控制 TAP 状态机的转换。通过 TMS 信号可以控制 TAP 在不同的状态间相互转换。TMS 信号在 TCK 的上升沿有效。TMS 在 IEEE 1149.1 标准里是强制要求的。

③ TDI (Test Data Input): TDI 是数据输入的接口。所有要输入到特定寄存器的数据都是通过 TDI 接口一位一位串行输入的（由 TCK 驱动）。TDI 在 IEEE 1149.1 标准里是强制要求的。

④ TDO (Test Data Output): TDO 是数据输出的接口。所有要从特定的寄存器中输出的数据都是通过 TDO 接口一位一位串行输出的（由 TCK 驱动）。TDO 在 IEEE 1149.1 标准里是强制要求的。

⑤ TRST: TRST 可以用来对 TAP Controller 进行复位(初始化)。不过这个信号接口在 IEEE 1149.1 标准里是可选的，并不是强制要求的。因为通过 TMS 也可以对 TAP Controller 进行复位(初始化)。

通过 TAP 接口，对数据寄存器 (DR) 进行访问的一般过程是：

① 通过指令寄存器 (IR)，选定一个需要访问的数据寄存器（也就是某条扫描链）。

② 把选定的数据寄存器连接到 TDI 和 TDO 之间。

③ 由 TCK 驱动，通过 TDI，把需要的数据输入到选定的数据寄存器中，同时把选定的数据寄存器中的数据通过 TDO 读出来。

TAP 的状态机如图 3-21 所示，总共有 16 个状态。在图中，每个六边形表示一个状态，六边形中标有该状态的名称和标识代码。图中的箭头表示了 TAP Controller 内部所有可能的状态转换流程。状态的转换是由 TMS 控制的，所以在每个箭头上标有 tms=0 或者 tms=1。在 TCK 的驱动下，从当前状态到下一个状态的转换是由 TMS 信号决定的。假设 TAP Controller 的当前状态为 Select-DR-Scan，在 TCK 的驱动下，如果 TMS=0，TAP Controller 进入 Capture-DR 状态；如果 TMS=1，TAP Controller 进入 Select-IR-Scan 状态。

观察图 3-21 可以发现，除了 Test-Logic Reset 和 Test-Run/Idle 状态外，其他的状态有些类似。例如，Select-DR-Scan 和 Select-IR-Scan 对应，Capture-DR 和 Capture-IR 对应，Shift-DR 和 Shift-IR 对应，等等。在这些对应的状态中，DR 表示 Data Register，IR 表示 Instruction Register。

如果需要捕获芯片某个管脚上的输出，首先需要把该管脚上的输出装载到边界扫描链的寄存器单元里去，然后通过 TDO 输出，这样我们就可以从 TDO 上得到相应管脚上的输出信号。如果要在芯片的某个管脚上加载一个特定的信号，则首先需要通过 TDI 把期望的信号移位到与相应管脚相连的边界扫描链的寄存器单元里去，然后把该寄存器单元的值加载到相应的芯片管脚。下面描述每个状态的具体含义。

① Test-Logic Reset: 系统上电后，TAP Controller 自动进入该状态。在该状态下，测试部分的逻辑电路全部被禁用，以保证芯片核心逻辑电路的正常工作。通过 TRST 信号也可以对测试逻辑电路进行复位，使得 TAP Controller 进入 Test-Logic Reset 状态。前面我们说过 TRST 是可选的一个信号接口，这是因为在 TMS 上连续加 5 个 TCK 脉冲宽度的“1”信号也可以对测试逻辑电路进行复位，

使得 TAP Controller 进入 Test-Logic Reset 状态。所以，在不提供 TRST 信号的情况下也不会产生影响。在该状态下，如果 TMS 一直保持为“1”，TAP Controller 将保持在 Test-Logic Rest 状态下；如果 TMS 由“1”变为“0”（TMS 总是在 TCK 的上升沿触发，下同），将使 TAP Controller 进入 Run-Test/Idle 状态。

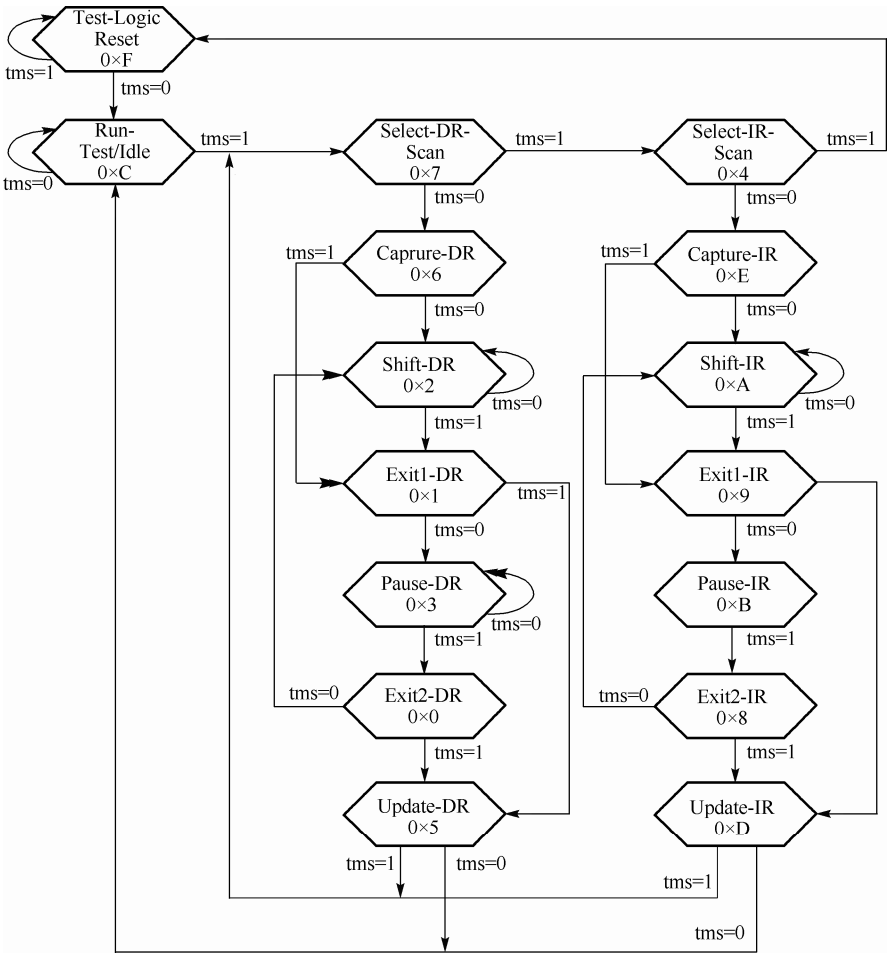


图 3-21 TAP Controller 状态机

② Run-Test/Idle: 这个是 TAP Controller 在不同操作间的一个中间状态。这个状态下的动作取决于当前指令寄存器中的指令。有些指令会在该状态下执行一定的操作，而有些指令在该状态下不需要执行任何操作。在该状态下，如果 TMS 一直保持为“0”，TAP Controller 将一直保持在 Run-Test/Idle 状态下；如果 TMS 由“0”变为“1”，将使 TAP Controller 进入 Select-DR-Scan 状态。

③ Select-DR-Scan: 这是一个临时的中间状态。如果 TMS 为“0”，TAP Controller 进入 Capture-DR 状态，后续的系列动作都将以数据寄存器作为操作对象；如果 TMS 为“1”，TAP Controller 进入 Select-IR-Scan 状态。

④ Capture-DR: 若 TAP Controller 在这个状态中，在 TCK 的上升沿，芯片输出管脚上的信号将被“捕获”到与之对应的数据寄存器的各个单元中，如果 TMS 为“0”，TAP Controller 进入 Shift-DR 状态；如果 TMS 为“1”，TAP Controller 进入 Exit-DR 状态。

⑤ Shift-DR: 在这个状态中, 由 TCK 驱动, 每一个时钟周期, 被连接在 TDI 和 TDO 之间的数据寄存器将从 TDI 接收一位数据, 同时通过 TDO 输出一位数据。如果 TMS 为“0”, TAP Controller 保持在 Shift-DR 状态; 如果 TMS 为“1”, TAP Controller 进入到 Exit1-DR 状态。假设当前的数据寄存器的长度为 4, 如果 TMS 保持为 0, 则在 4 个 TCK 时钟周期后, 该数据寄存器中原来的 4 位数据(一般是在 Capture-DR 状态中捕获的数据)将从 TDO 输出来; 同时该数据寄存器中的每个寄存器单元中将分别获得从 TDI 输入的 4 位新数据。

⑥ Update-DR: 在 Update-DR 状态下, 由 TCK 上升沿驱动, 数据寄存器中的数据将被加载到相应的芯片管脚上去, 用以驱动芯片。在该状态下, 如果 TMS 为“0”, TAP Controller 将回到 Run-Test/Idle 状态; 如果 TMS 为“1”, TAP Controller 将进入 Select-DR-Scan 状态。

⑦ Select-IR-Scan: 这是一个临时的中间状态。如果 TMS 为“0”, TAP Controller 进入 Capture-IR 状态, 后续的系列动作都将以指令寄存器作为操作对象; 如果 TMS 为“1”, TAP Controller 进入 Test-Logic Reset 状态。

⑧ Capture-IR: 若 TAP Controller 在这个状态中, 在 TCK 的上升沿, 一个特定的逻辑序列将被装载到指令寄存器中。如果 TMS 为“0”, TAP Controller 进入 Shift-IR 状态; 如果 TMS 为“1”, TAP Controller 进入 Exit-IR 状态。

⑨ Shift-IR: 在这个状态中, 由 TCK 驱动, 每一个时钟周期, 被连接在 TDI 和 TDO 之间的指令寄存器将从 TDI 接收一位数据, 同时通过 TDO 输出一位数据。如果 TMS 为“0”, TAP Controller 保持在 Shift-IR 状态; 如果 TMS 为“1”, TAP Controller 进入到 Exit-IR 状态。假设指令寄存器的长度为 4, 如果 TMS 保持为 0, 则在 4 个 TCK 时钟周期后, 指令寄存器中原来的 4 位长的特定逻辑序列(在 Capture-IR 状态中捕获的特定逻辑序列)将从 TDO 输出来, 该特定的逻辑序列可以用来判断操作是否正确; 同时指令寄存器将获得从 TDI 输入的一个 4 位长的新指令。

⑩ 在这个状态中, 在 Shift-IR 状态下输入的新指令将被用来更新指令寄存器。

访问指令寄存器的一般过程如下: 系统上电, TAP Controller 进入 Test-Logic Reset 状态, 然后依次进入: Run-Test/Idle→Select-DR-Scan→Select-IR-Scan→Capture-IR→Shift-IR→Exit1-IR→Update-IR, 最后回到 Run-Test/Idle 状态。在 Capture-IR 状态中, 一个特定的逻辑序列被加载到指令寄存器中, 然后进入到 Shift-IR 状态。在 Shift-IR 状态下, 通过 TCK 的驱动, 可以将一条特定的指令送到指令寄存器中。每条指令都将确定一条相关的数据寄存器。然后从 Shift-IR→Exit1-IR→Update-IR。在 Update-IR 状态, 刚才输入到指令寄存器中的指令将用来更新指令寄存器。最后, 进入到 Run-Test/Idle 状态, 指令生效, 完成对指令寄存器的访问。

访问数据寄存器的一般过程如下: 当前可以访问的数据寄存器由指令寄存器中的当前指令决定。要访问由刚才的指令选定的数据寄存器, 需要以 Run-Test/Idle 为起点, 依次进入 Select-DR-Scan→Capture-DR→Shift-DR→Exit1-DR→Update-DR, 最后回到 Run-Test/Idle 状态。在这个过程中, 被当前指令选定的数据寄存器会被连接在 TDI 和 TDO 之间。通过 TDI 和 TDO, 就可以将新的数据加载到数据寄存器中, 同时也可以捕获数据寄存器中的数据。具体过程如下。在 Capture-DR 状态中, 由 TCK 的驱动, 芯片管脚上的输出信号会被“捕获”到相应的边界扫描寄存器单元中。这样, 当前的数据寄存器当中就记录了芯片相应管脚上的输出信号。接下来从 Capture-DR 进入到 Shift-DR 状态中。在 Shift-DR 状态中, 由 TCK 驱动, 在每一个时钟周期内, 一位新的数据可以通过 TDI 串行输入到数据寄存器中, 同时数据寄存器可以通过 TDO 串行输出一位先前捕获的数据。在经过与数据寄存器长度相同的时钟周期后, 就可以完成新信号的输入和捕获数据的输出。接下来通过 Exit1-DR 状态进入到 Update-DR 状态。在 Update-DR 状态中, 数据寄存器中的新数据被加

载到与数据寄存器的每个寄存器单元相连的芯片管脚上去。最后，回到 Run-Test/Idle 状态，完成对数据寄存器的访问。

下面来看一个例子。假设 TAP Controller 现在处于 Run-Test/Idle 状态，指令寄存器中已经成功地写入了一条新的指令，该指令选定的是一条长度为 6 的边界扫描链。下面让我们来看看实际如何来访问这条边界扫描链。图 3-22 所示是测试芯片及其被当前指令选定的长度为 6 的边界扫描链。由图 3-22 可以看出，当前选择的边界扫描链由 6 个边界扫描移位寄存器单元组成，并且被连接在 TDI 和 TDO 之间。TCK 时钟信号与每个边界扫描移位寄存器单元相连。每个时钟周期可以驱动边界扫描链的数据由 TDI 到 TDO 的方向移动一位，这样，新的数据可以通过 TDI 输入一位，边界扫描链的数据可以通过 TDO 输出一位。经过 6 个时钟周期，就可以完全更新边界扫描链中的数据，而且可以将边界扫描链里捕获的 6 位数据通过 TDO 全部移出来。

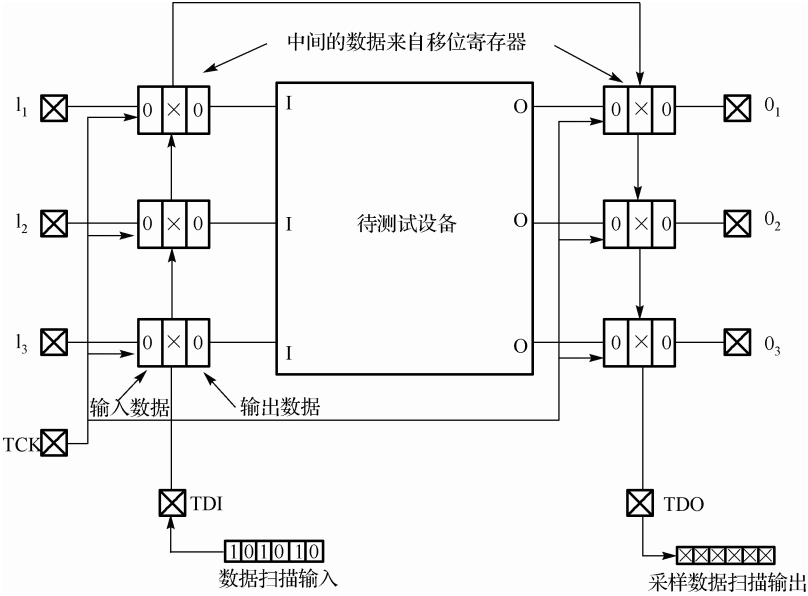


图 3-22 测试芯片及其当前选定的边界扫描链

3. 指令寄存器、公共指令以及数据寄存器

在 IEEE 1149.1 标准中，规定了一些指令寄存器、公共指令和相关的一些数据寄存器。对于特定的芯片而言，芯片厂商一般都会在 IEEE 1149.1 标准的基础上扩充一些私有的指令和数据寄存器，以帮助在开发过程中进行测试和调试。下面简单介绍 IEEE 1149.1 规定的一些常用指令及其相关的寄存器。与 ARM 相关的私有指令和寄存器将在后面专门介绍。

- ① 指令寄存器：指令寄存器运行被装载的特定指令，用来选择需要执行的测试，或者选择需要访问的测试数据寄存器。每个支持 JTAG 调试的芯片必须包含一个指令寄存器。
- ② BYPASS 指令和 Bypass 寄存器：Bypass 寄存器是一个一位的移位寄存器，通过 BYPASS 指令，可以将 Bypass 寄存器连接到 TDI 和 TDO 之间。在不需要进行任何测试时，将 Bypass 寄存器连接在 TDI 和 TDO 之间，在 TDI 和 TDO 之间提供一条长度最短的串行路径。这样允许测试数据可以快速通过当前的芯片并送到开发板上别的芯片上去。
- ③ IDCODE 指令和 Device Identification 寄存器：Device Identification 寄存器中可以包括生产厂商的信息、部件号码和器件的版本信息等。使用 IDCODE 指令，就可以通过 TAP 来确定器件的这些相

关信息。例如，ARM multi-ICE 可以自动识别当前调试的是什么芯片，其实就是通过 IDCODE 指令访问 Device Identification 寄存器来获取的。

④ INTEST 指令和 Boundary-Scan 寄存器: Boundary-Scan 寄存器就是前面例子中说到的边界扫描链。通过边界扫描链，可以进行部件间的连通性测试。当然，更重要的是可以对测试器件的输入/输出进行观测和控制，以达到测试器件的内部逻辑的目的。INTEST 指令是在 IEEE 1149.1 标准中定义的一条很重要的指令，结合边界扫描链，该指令允许对开发板上器件的系统逻辑进行内部测试。在 ARM JTAG 调试中，这是一条频繁使用的测试指令。

寄存器分为两大类：指令寄存器和数据寄存器。上面提到的 Bypass 寄存器、DeviceIdentification 寄存器和 Boundary-scan 寄存器（边界扫描链）都属于数据寄存器。在调试中，边界扫描寄存器（边界扫描链）最重要，使用也最频繁。

3.3.2 基于 JTAG 的片上在线仿真的系统结构

基于 JTAG 的片上在线仿真调试系统结构如图 3-23 所示，它包括 3 部分：位于主机上的调试器 (Debugger)，例如 ARM 公司的 AXD 等，通常调试器是运行于主机的集成开发环境 (IDE) 上的一个软件；包括片上在线仿真调试逻辑的目标系统；在主机和目标系统之间进行协议分析、转换的模块。下面分别介绍这些组成部分。

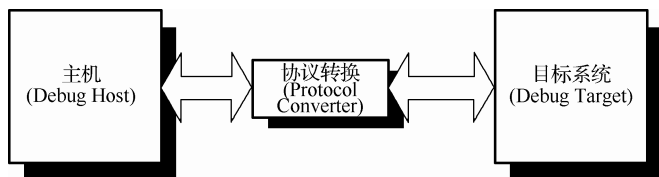


图 3-23 基于 JTAG 的片上在线仿真调试系统结构图

位于主机上的调试器主要用来接收用户的命令，并将其发送到目标系统中的调试部件，接收从目标系统返回的数据，并以一定格式显示给用户。ARM 公司的 AXD 就是一个基于 Windows 操作系统的调试器。

目标机以 ARM7TDMI 为例，该 CPU 集成了三条基本的扫描链 0、1 和 2，分别用于测试、仿真和对嵌入式 ICE 模块的编程。目标机的结构如图 3-24 所示。它主要包括下面 3 部分：

- ① 需要进行调试的 CPU 内核。
- ② Embedded ICE 逻辑电路，包括一组寄存器和比较器，它可以用来产生调试时需要的异常，如产生断点等。这部分电路实际上就是集成在片内的在线仿真 (ICE) 逻辑。
- ③ TAP 控制器可以通过 JTAG 接口控制各个硬件扫描链。

目标机包含的硬件调试功能扩展部件可以实现下面的功能：

- 停止目标程序的执行。
- 查看目标内核的状态。
- 查看和修改存储器的内容。
- 继续程序的执行。

图 3-24 中 3 条扫描链的含义如下。

- ① 扫描链 0：可以用来访问 ARM7TDMI CPU 核

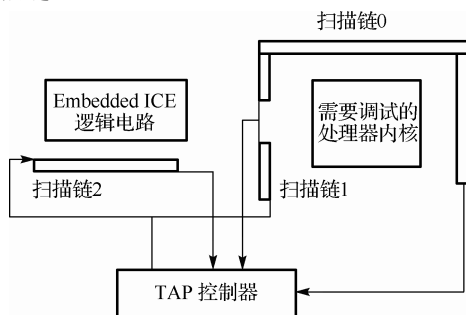


图 3-24 被调试的目标系统结构

的所有输入/输出信号，主要用于设备间测试（EXTEST）和核心内部测试（INTEST），整个扫描链从输入到输出包含下面几部分：数据总线从位 0 到位 31；控制信号；地址总线从位 0 到位 31。

② 扫描链 1：该扫描链的扫描单元共 33 个，为扫描链 0 的一个子集，包括数据总线的 0~31 位和一个 DBGBREAK 信号。DBGBREAK 信号在进入断点时可以标示 CPU 核心是因为指令断点进入仿真状态还是因为数据断点进入仿真状态。而在仿真过程中，通过主机设置 DBGBREAK 信号则可以告诉核心 CPU 是否同步系统时钟执行指令（在仿真状态下，要想得到系统状态以系统时钟执行指令），低表示核心以调试速度（也就是以测试时钟驱动）执行，高表示以正常速度执行。访问数据总线的 32 个扫描单元可以在仿真状态下把核心和存储器系统分离，核心访问的数据都是通过扫描单元送来的，核心要输出的数据也是通过扫描单元移出送给仿真主机的。因此在仿真状态下可以把指令插入到核心的指令流中，从而执行一些调试所需要的处理器指令。

③ 扫描链 2：主要用于访问 Embedded ICE 逻辑部件中的各个寄存器。

位于主机和目标系统之间的协议转换器完成主机和系统之间的信息沟通，实现了 PC 协议与 JTAG 协议的协议转换功能（通常情况下，PC 没有专门用于调试的通信接口，而往往是通过串口、并口、USB 口或者以太网口与目标系统通信，这就必须实现将 PC 通信接口协议转换为 JTAG 协议的协议转换器）。通常它是一个独立硬件模块，与主机之间通过串行口或者并行口连接，与目标机之间通过 JTAG 接口相连。需要说明的是，通常情况下人们往往将协议转换器称为仿真器，这其实是一种误解或者商业策略（甚至 ARM 公司也将自己推出的协议转换器称为 MultiICE），读者必须非常清楚真正的仿真器已经被集成到被调试芯片中去了。

3.3.3* ARM7TDMI 内核调试原理

本节详细介绍 ARM7TDMI 的调试原理，例如，如何设置断点？如何让 ARM7TDMI 进入调试状态？在调试状态下，如何访问 ARM7TDMI 的内部寄存器？如何访问内存单元？

我们先来看看 ARM7TDMI 的正常运行状态和调试状态的区别。在正常运行状态下，ARM7TDMI 由 MCLK（Memory Clock）驱动。在调试状态下，ARM7TDMI 的正常运行被打断，并且和系统的其他部分隔离开来。在调试状态下，我们可以通过扫描链插入特定的 ARM 指令来读写 ARM7TDMI 的内部寄存器和修改内存的内容。在调试状态下，ARM7TDMI 由内部的调试时钟 DCLK（Debug Clock）驱动。DCLK 比 MCLK 慢很多，所以在调试状态下，插入的 ARM 指令的运行速度会比较慢。在完成需要的操作后，可以用 RESTART JTAG 指令让 ARM7TDMI 返回到正常运行状态，恢复原来的运行状态。

1. ARM7TDMI 的 JTAG 指令寄存器及常用 JTAG 命令

ARM7TDMI 的 JTAG 指令寄存器长度是 4 位，通过 TAP 和 JTAG 接口，可以把指令装载到指令寄存器中。在 CAPTURE-IR 状态下，固定值 B0001 总是被装载到指令寄存器中。在 SHIFT-IR 状态中，可以把 ARM7TDMI 支持的新指令从 TDI 串行输入，同时固定值 B0001 会从 TDO 串行输出。通过这个输出的固定值，可以判断当前的操作是否正确。在 UPDATE-IR 状态，新输入的指令被装载到指令寄存器中。最后回到 Run-Test/Idle 状态后，新指令（注意，这里的指令是 JTAG 指令，而不是 ARM CPU 的指令）立即生效。

下面先来看看在 ARM7TDMI 调试中经常用到的几条 JTAG 指令。

① IDCODE：该指令的二进制代码是 B1110。IDCODE 指令将 Device Identification Code 寄存器连接到 TDI 和 TDO 之间。Device Identification Code 寄存器的长度是 32 位，通过 TAP 就可以将 ARM7TDMI 的 ID 读出来。ARM7TDMI 的 ID 是 0x1F0F0F0F。在 Capture-DR 状态下，器件的 ID 被捕获到 ID 寄

寄存器中。在 Shift-DR 状态下, 先前捕获的 ID 通过 TDO 位移出来, 总共需要 32 个 TCK 时钟周期。在 Update-DR 状态下, ID 寄存器不受任何影响。

② SCAN_N: 该指令的二进制代码是 B0010。ARM7TDMI 提供了 4 条扫描链, 通过 SCAN_N 指令可以选择需要访问的扫描链。一般来说, 需要访问 Embedded ICE-RT 的寄存器时, 选择扫描链 2; 需要插入指令到 ARM7TDMI 内核去执行的时候, 选择扫描链 1。选择扫描链的过程如下: 先把 SCAN_N 指令装载到指令寄存器中, 该指令会将长度为 4 位的扫描链选择寄存器连接到 TDI 和 TDO 之间; 然后进入到 Capture-DR 状态, 在这个状态下, 固定值 B1000 将被捕获到扫描链选择寄存器中; 在 Shift-DR 状态下, 将需要选择的扫描链的号码通过 TDI 输入到扫描链选择寄存器中; 在 Update-DR 状态下, 被选择的扫描链将被连接到 TDI 和 TDO 之间。在 TAP 被复位以后, 默认状态下选择的是扫描链 3。在使用 SCAN_N 选定了一条扫描链后, 当前选定的扫描链会一直保持不变, 直到下一次调用 SCAN_N 选择另外的扫描链。

③ BYPASS: 该指令的二进制代码是 B1111。BYPASS 指令将 1 位长的 Bypass 寄存器连接到 TDI 和 TDO 之间。

④ INTEST: 该指令的二进制代码是 B1100。INTEST 指令将通过 SCAN_N 选定的扫描链置于内部测试模式。

⑤ RESTART: 该指令的二进制代码是 B0100。RESTART 指令用来使 ARM7TDMI 处理器从调试状态回到正常的运行状态。

2. Embedded ICE 模块

要支持基于 JTAG 的调试, 在 SoC 芯片需要集成 Embedded ICE 硬件模块。调试主机也需要运行一个调试器程序。目标系统上的 Embedded ICE 电路主要功能是在断点处向处理器内核发出调试请求, 使处理器进入调试状态。Embedded ICE 电路是通过判断处理器的地址/数据总线来确定是否该向处理器发送调试请求。例如, 用户在一个指令处设了程序断点, 处理器去取这条指令时发出指令地址, ICE 将该地址和断点指令处地址进行比较, 如果相等就在这条指令上加上标识, 当处理器运行这条指令时 CPU 就会进入调试状态。进入调试状态后, 处理器的状态(寄存器值)可以通过边界扫描电路而获得。这种边界扫描方法需要增加扫描电路, 但是不占用系统的 ROM 和 RAM。

Embedded ICE 调试方法用得比较普遍, ARM 公司的 ARM7、ARM9 系列处理器核和 Motorola 公司的 EZ、VZ、SZ 系列等都在处理器里集成了 Embedded ICE 电路。在这种调试架构下, 除了可以完成驻留监控软件方式所能完成的所有调试功能外, 还可以支持硬件断点、数据断点、硬件方式的单步执行指令等多种调试类型。

在 ARM7TDMI 处理器中, Embedded ICE 逻辑部件提供了集成在芯片内的对内核进行调试的功能。这部分功能是通过处理器上的 TAP 控制器串行控制的。图 3-25 表示了处理器内核、Embedded ICE 逻辑部件以及 TAP 控制器之间的关系, 以及一些主要的控制信号。

Embedded ICE 逻辑部件包含下面几部分:

- 两个数据断点(watchpoint)寄存器。
- 两个独立的寄存器: 调试控制寄存器和调试状态寄存器。
- 调试通信通道(DCC)。

两个数据断点寄存器可以被用来设置数据断点或者程序断点。当设置程序断点时, 若当前 CPU 发出的地址与数据断点寄存器的值相等, Embedded ICE 逻辑部件停止程序的执行。当设置数据断点时, 若 CPU 发出的访问数据的地址与数据断点寄存器的值相等, Embedded ICE 逻辑部件停止程序的执行。与驻留监控软件调试方式不同, 这时程序断点可以设置在 ROM 中, 这是因为 Embedded ICE 逻辑部件

提供了需要的硬件支持，而硬件指令断点实际上就是比较地址总线值。在数据断点寄存器中的数据中的位可以被屏蔽，使其在进行比较时不起作用，从而使得断点的设置更为灵活。

调试通信通道（DCC）用来在主机上的调试器和目标处理器之间建立通信信道。在 ARM7TDMI 中，它是作为一个协处理器来实现的。它包括：

- 一个 32 位的通信数据读寄存器。
- 一个 32 位的通信数据写寄存器
- 一个 6 位的通信控制寄存器。

通过这些接口，DCC 可以在主机上的调试器和目标处理器之间建立通信信道。在所有调试信号中，下面 3 个是最为主要的：

- BREAKPT 产生指令断点或者数据断点。
- DBGRQ 请求处理器进入调试状态。
- DBGACK 表明处理器已经进入调试状态

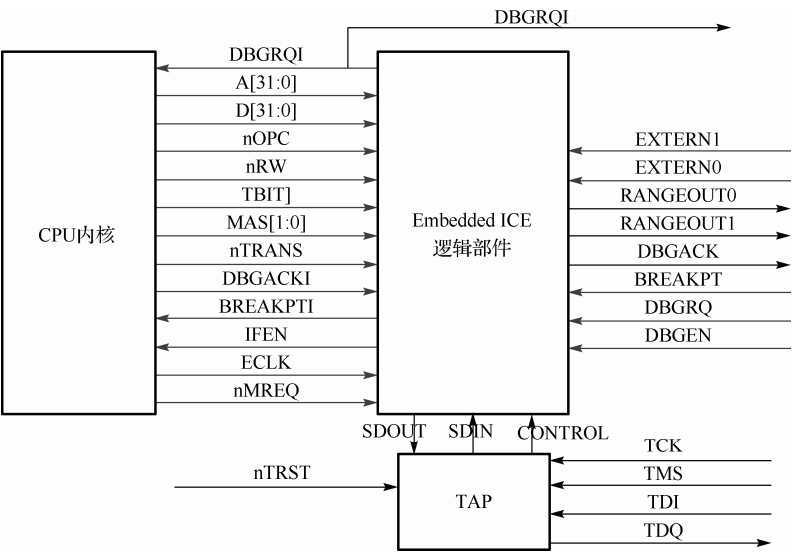


图 3-25 CPU 内核、Embedded ICE 逻辑部件以及 TAP 控制器之间的关系

ICE 模块最主要的功能就是产生 BREAKPT 和 DBGRQ 信号并送给内核，当发生程序断点或数据断点时则 BREAKPT 信号有效，当有调试请求时则 DBGRQ 信号有效。这两个信号的产生是通过比较数据总线、地址总线、控制总线上的值和对应的断点寄存器的值产生的。BREAKPT 信号的产生原理如图 3-26 所示。ICE 模块共有两组断点寄存器，每组断点寄存器都包括一个 32 位的数据寄存器 and 对应的屏蔽寄存器、一个 32 位的地址寄存器和对应的屏蔽寄存器、一个 9 位的控制寄存器和对应的 8 位屏蔽寄存器。数据寄存器的值用于和数据总线上的 32 位数据进行比较，数据屏蔽寄存器可以决定比较 32 位数据中的哪几位。地址寄存器的值用于和地址总线上的 32 位地址进行比较，地址屏蔽寄存器可以决定比较 32 位地址中的哪几位。控制寄存器的值用于和内核输出的一些控制信号做比较，控制屏蔽寄存器决定比较哪几个控制信号。

需要设置在某地址处的指令执行时触发断点情况下各寄存器配置如下：配置地址寄存器为硬件断点指令的地址，屏蔽地址寄存器设置为仅屏蔽地址总线的低两位（ARM 指令是 32 位对齐的）；不比较数据总线，屏蔽数据寄存器的值设置为屏蔽所有数据位。图 3-26 中的电路可以配置一个指令断点，指令断点可以设置在 RAM 中或者 ROM 中。

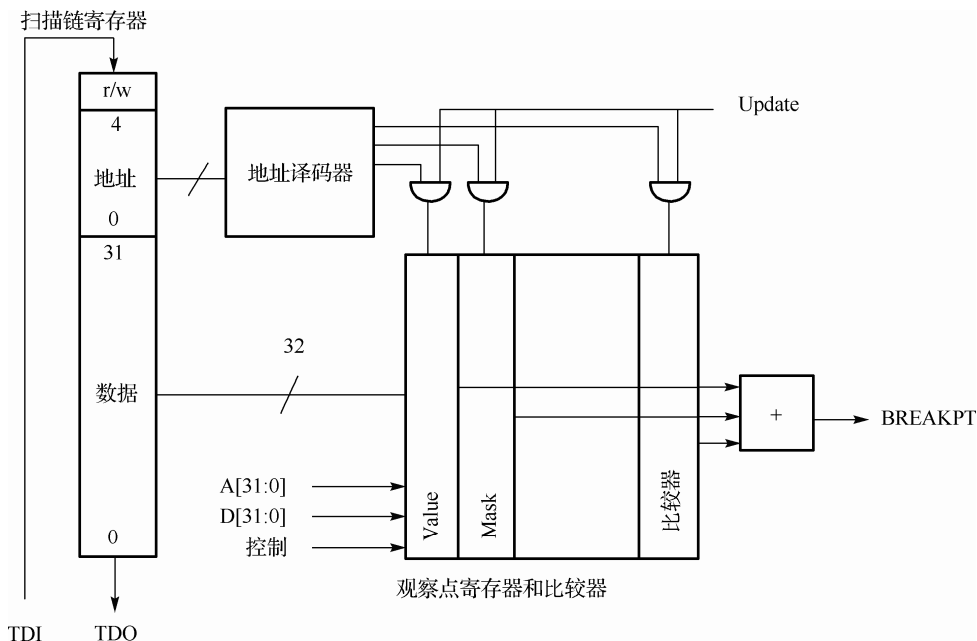


图 3-26 BREAKPT 信号的产生原理

需要设置在某地址处的数据被访问时触发断点情况下各寄存器配置如下：配置地址寄存器为断点数据的地址，屏蔽地址寄存器设置为不屏蔽所有地址；不比较数据总线，屏蔽数据寄存器的值设置为屏蔽所有数据位。图 3-26 中的电路可以配置一个数据断点，数据断点可以设置在 RAM 中或者 ROM 中。

需要设置数据总线上的数据为某条特定的指令触发断点时各寄存器配置如下：不比较地址总线，屏蔽地址寄存器设置为屏蔽所有地址；数据寄存器设置为特定的指令，屏蔽数据寄存器的值设置为不屏蔽所有数据位。

内核在指令断点、数据断点和有调试请求的情况下（即嵌入式 ICE 模块的输出信号 BREAKPT 或 DBGCRQ 有效）使 DBGACK 信号有效，进入仿真状态。当内核检测到 BREAKPT 信号有效后，如果是由于指令断点使得 BREAKPT 信号有效，则对这条断点指令做好标记，当断点指令到达了指令流水线的执行阶段后，内核进入仿真状态，断点指令之前的指令都执行完毕，断点指令虽然进入了执行阶段但并不改变寄存器的状态，所以从仿真状态返回时应把断点指令的标记清除然后从断点指令处开始重新执行程序。如果是由于数据断点使得 BREAKPT 信号有效，则引起数据断点的指令执行完后进入仿真状态，所以从仿真状态退出时应返回到引起数据断点的指令的下一条指令处开始执行。如果有外部调试请求而 DBGCRQ 信号有效，内核执行完当前指令后进入仿真状态。内核进入仿真状态后，DBGACK 信号为高，内核时钟会由原来的主时钟切换为由 JTAG 接口的 TCK 产生的时钟 DCLK。

内核进入仿真状态后，可以通过扫描链 1 的 BREAKPT 位告诉仿真主机内核是由于指令断点还是由于数据断点进入仿真状态的。在仿真状态下，TAP 控制器的指令寄存器中装入 INTEST 指令，选择扫描链 1 作为当前扫描链，接下来调试器程序可以通过把内核寄存器存储指令（STM 等）装入扫描链 1，进而插入到内核的指令流中，把内核寄存器的值扫描出来。内核是以仿真速度（以 DCLK 作为执行指令的时钟）执行 STM 指令的，这个 DCLK 是非常慢的，因为要通过 JTAG 接口的 TCK 时钟把 32 位的指令经过 32 个 TCK 时钟移入扫描链，然后产生一个 DCLK 时钟，之后又要通过 TCK 时钟把扫描链 1 的内核寄存器的值一位一位地移出来。

如何在仿真状态下观测内核以外的系统状态（如读存储器的内容等）呢？因为有些存储器（如 SDRAM 等）只能在系统时钟 MCLK 下工作，所以内核就不能以仿真速度执行指令访问存储器了，而要以系统速度执行指令，解决这个问题的方法是让内核在执行存储器访问命令时重新同步回系统时钟 MCLK。我们把存储器访问指令之前的那条指令装入扫描链 1 的同时使 DBGBREAK 位为高，执行完成后把存储器访问命令装入扫描链 1 同时使 DBGBREAK 位为低，把这条指令放到数据总线上以后，在 TAP 控制器中选择 BYPASS 指令，这会使得内核重新同步回系统时钟 MCLK，以 MCLK 时钟执行这条存储器访问命令，执行完命令后又切换回 DCLK 时钟，此时在 TAP 控制器中选择 INTEST 指令，则又重新回到仿真状态。在仿真状态下以系统速度执行指令时 DBGACK 信号为低。

从仿真状态退出包括恢复内核的内部状态、执行一条跳转到进入仿真状态之前未执行的指令、然后同步回系统时钟 MCLK。把最后一条跳转指令之前的那条指令装入扫描链 1 时使 DBGBREAK 位为高，执行完毕后把跳转指令装入扫描链 1 同时使 DBGBREAK 位为低，把这条指令放到数据总线上以后，在 TAP 控制器中选择 RESTART 指令，当状态机进入运行-测试/空闲模式时，扫描链运行于系统模式，内核重新同步回系统时钟 MCLK，进行正常的指令执行。

下面举例说明如何在调试状态下访问 ARM7TDMI 的内部通用寄存器和系统内存。

3. 通过扫描链访问处理器内核的寄存器

1) 读取通用寄存器 R0 的值

要读取寄存器 R0 的值，可以用指令 STR R0, [R0]来实现。该指令将寄存器 R0 的值存储到内存单元地址为 R0 的存储器中。因为 ARM7TDMI 处于调试状态的时候，ARM7TDMI 和外部是隔离开来的，所以该指令实际上不能访问真实的内存单元，也不会对内存单元产生任何影响。使用指令 STR R0, [R0]的目的是使得寄存器 R0 的值出现在数据总线上，这样我们就可以通过扫描链 1 将其捕获，然后从 TDO 输出。指令 STR R0, [R0]的执行需要两个指令周期（虽然绝大多数的 ARM7TDMI 指令可以在一个周期内完成，也就是在三级流水中的 E 级，但是对于 LDR/STR 指令的执行，处理器在 E 级计算目的地址，然后在其后增加一个 M 周期，用于访存操作，关于 ARM7TDMI 的流水线，读者可以参阅第 4 章 4.2.2 节）。在第 1 个指令执行周期，执行地址计算；在第 2 个指令执行周期，将寄存器 R0 的值放到数据总线上。要读取寄存器 R0 的值，对边界扫描链 1 的操作过程如下（在下面描述的每一个步骤，TAP 状态都是从 Run-Test/Idle 状态触发，最后回到 Run-Test/Idle 状态）：

① 插入指令 STR R0, [R0] & BREAKPT=0。这一步相当于指令 STR R0, [R0]的取指令周期（3 级流水线）。

② 插入空指令 MOV R0, R0 & BREAKPT=0。这一步读取新指令 MOV R0, R0，同时，相当于指令 STR R0, [R0]的译码指令周期（3 级流水线）。

③ 插入空指令 MOV R0, R0 & BREAKPT=0。这一步读取新指令 MOV R0, R0，同时，插入的 STR R0, [R0]指令开始执行（3 级流水线）。在这一步，STR R0, [R0]指令处在第 1 个指令执行周期，在该周期，先执行地址计算。

④ 通过扫描链 1 读出捕获的数据总线上的数据。指令 STR R0, [R0]继续执行该指令的第 2 个指令周期。寄存器 R0 的值会被放到数据总线上。同时，指令 STR R0, [R0]执行完毕。这时，从扫描链中读出的数据就是寄存器 R0 的值。另外，因为还处在 STR R0, [R0]的指令执行周期，所以访问扫描链 1 的时候，通过 TDI 输入的空指令实际上会被 ARM7TDMI 忽略。

⑤ 至此，我们成功获得了寄存器 R0 的值。因为除了 STR R0, [R0]指令外，插入的另外两条指令都是空指令，所以后面可以不考虑这两条指令的执行结果。

2) 修改通用寄存器 R0 的值

要修改寄存器 R0 的值, 可以用指令 LDR R0, [R0]来实现。该指令将内存单元地址为 R0 的值存储到寄存器 R0 中去。因为在 ARM7TDMI 处于调试状态的时候, ARM7TDMI 和外部是隔离开来的, 所以该指令实际上不能访问内存单元, 也不会对内存单元产生任何影响。使用指令 LDR R0, [R0]的目的是: 该指令要从内存单元中取数据放到数据总线上, 并写入 R0 寄存器。这样我们就可以通过扫描链 1 将需要写入的值放到数据总线上, 达到修改寄存器 R0 的目的。指令 LDR R0, [R0]的执行需要 3 个指令执行周期(流水线的 E 级负责计算目标地址, CPU 还将插入 M 周期负责存储器访问, 并在 M 周期后插入 W 周期, 将读取的数据写回寄存器堆)。在第 1 个指令执行周期, 执行地址计算; 在第 2 个指令执行周期, 从内存单元 R0 中读取数据, 并将数据放到数据总线上; 在第 3 个指令执行周期, 将数据总线上的数据写到寄存器 R0 中去。要修改寄存器 R0 的值, 对边界扫描链 1 的操作过程如下(在下面描述的每一个步骤, TAP 状态都是从 Run-Test/Idle 状态触发, 最后回到 Run-Test/Idle 状态):

① 插入指令 LDR R0, [R0] & BREAKPT=0。这一步相当于指令 LDR R0, [R0]的取指令周期(3 级流水线)。

② 插入空指令 MOV R0, R0 & BREAKPT=0。这一步读取新指令 MOV R0, R0, 同时, 相当于指令 LDR R0, [R0]的译码指令周期(3 级流水线)。

③ 插入空指令 MOV R0, R0 & BREAKPT=0。这一步读取新指令 MOV R0, R0, 同时, 插入的 LDR R0, [R0]指令开始执行(3 级流水线)。在这一步, LDR R0, [R0]指令处在第 1 个指令执行周期, 在该周期, 先执行地址计算。

④ 通过扫描链 1 读出捕获的数据总线上的数据。指令 LDR R0, [R0]继续执行该指令的第 2 个指令周期。通过扫描链 1, 将需要写入的新值放到数据总线上。对 ARM7TDMI 而言, 相当于从系统内存中取得了所需的数据。

⑤ 插入空指令 MOV R0, R0 & BREAKPT=0。指令 LDR R0, [R0]继续执行第 3 个指令周期。数据总线上的数据写到寄存器 R0 中去, 完成对寄存器 R0 的修改, 指令 LDR R0, [R0]执行完毕。另外, 在这一步, 因为还处在 LDR R0, [R0]的指令执行周期, 所以访问扫描链 1 的时候, 通过 TDI 输入的空指令实际上会被 ARM7TDMI 忽略。

⑥ 至此, 我们成功修改了寄存器 R0 的值。因为除了 LDR R0, [R0]指令外, 插入的另外 3 条指令都是空指令, 所以后面我们可以不考虑这 3 两条指令的执行结果。

4. 通过扫描链访问存储器

1) 读内存

前面说过, 当 ARM7TDMI 处于调试状态的时候, 不能访问系统的存储空间, 这是因为在调试状态下, 指令的执行是由 DCLK 驱动的。而对存储空间(内存)的访问需要 MCLK 的驱动, 那么在调试状态下如何实现对存储空间的访问呢? 这个时候, 边界扫描链 1 的 BREAKPT 位就派上用场了。在通过扫描链 1 插入一条指令的时候, 如果将 BREAKPT 位置 1, 意味着这条指令的下面一条指令将在 MCLK 的驱动下执行, 执行完毕后自动返回调试状态。这样, 通过扫描链 1 的 BREAKPT 位就可以实现对系统存储空间的访问了。让我们来看看要读取内存地址 ADDR 上的 32 位数据的大概步骤。首先, 我们将要访问的地址 ADDR 写到寄存器 R0 中; 然后用指令 LDR R1, [R0]将地址 ADDR 处 32 位长的数据复制到寄存器 R1 中; 最后, 将寄存器 R1 的值读出来, 就得到了存储空间地址 ADDR 处的值。修改寄存器 R0 的值和读取寄存器 R1 的值的步骤在前面都详细介绍了, 下面介绍 BREAKPT 如何用指令 LDR R1, [R0]来真正地访问系统存储空间, 将地址 R0 处的内容复制到寄存器 R1 中。具

体过程如下（在下面描述的每一个步骤，TAP 状态都是从 Run-Test/Idle 状态触发，最后回到 Run-Test/Idle 状态）：

① 插入空指令 MOV R0, R0 & BREAKPT=1。首先插入一条空指令，该指令的主要目的是将扫描链 1 的 BREAKPT 信号置 1。这样，这条空指令的下一条指令将在 MCLK 的驱动下回到正常状态下运行。

② 插入指令 LDR R1, [R0] & BREAKPT=0。因为上一条指令将扫描链 1 的 BREAKPT 置 1 了，所以，这条指令将在 MCLK 的驱动下回到正常的系统状态下去执行。

③ 将 JTAG RESTART 指令插入到 ARM7TDMI 的 JTAG 指令寄存器中。将 RESTART 插入到 JTAG 指令寄存器中有几个作用。一是使 ARM7TDMI 重新同步于 MCLK 信号；二是使刚才插入的指令 LDR R1, [R0] 在系统正常状态下，在 MCLK 的驱动下执行；三是在系统正常状态下执行完指令 LDR R1, [R0] 后，使 ARM7TDMI 自动返回到调试状态下。

④ 在将 RESTART 写入到 JTAG 指令寄存器，并且在 TAP Controller 回到 Run-Test/Idle 状态后，ARM7TDMI 将会暂时返回到正常的运行状态。在正常的运行状态下，ARM7TDMI 将以系统速度（MCLK 驱动）完成指令 LDR R1, [R0] 的执行。执行完毕后，ARM7TDMI 将自动返回到调试状态。这样，寄存器 R1 中就已经有了地址 ADDR 处数值的一个副本了。最后需要做的就是通过前面介绍的方法，在调试状态下，将 R1 的值读出来。在需要访问内存的时候，需要注意的就是扫描链 1 中 BREAKPT 位的设置，以及 JTAG RESTART 指令的使用。

2) 写内存

修改内存的过程和读取内存的过程类似，大概步骤如下：将内存地址 ADDR 写到寄存器 R0 中，将新的值写到寄存器 R1 中，最后利用指令 STR R1, [R0] 完成实际的内存访问。在将指令 STR R1, [R0] 插入到扫描链 0 之前，将扫描链 1 的 BREAKPT 位置 1，以使得关键指令 STR R1, [R0] 能在 MCLK 的驱动下回到正常的系统状态下去执行，以便能正常访问系统存储。

上面介绍了通过扫描链 1 如何访问 ARM7TDMI 的通用寄存器和系统存储空间。为了简单起见，我们使用了最简单的 LDR 和 STR 指令。在实际调试过程中，可以使用 LDM 和 STM 指令，这样一次可以读写多个寄存器或者是多个地址上的数据，提高效率。从上面的讨论可以看出，在调试状态下，每次对通用寄存器和系统存储系统进行访问都需用多条指令来实现。每次插入一条指令到扫描链 1 中去，都需要 33 个 TCK 时钟周期。显然，采用 JTAG 通过串行的方式进行数据的输入/输出，其效率是比较低的。好在 TCK 的时钟频率通常在 10~20MHz 之间，甚至更高（比如 ARM 公司最新的 Dstream 调试与跟踪单元支持 60MHz 的 TCK。JTAG TCK 时钟频率越高，对于 PCB 板级布线的要求也就越高），所以基本可以满足调试的需要。

3.4 ARM 的集成开发环境

ARM 公司推出的软件集成开发工具（IDE）经过了几代产品系列的演进。早期比较成功和广受欢迎的开发套件称为 ADS，后来该产品的升级版被称为 RVDS（RealView Developer Suit）。为了配合 ARM 公司向控制器领域的延伸，该公司于 2005 年收购了大名鼎鼎的单片机开发工具提供商 Keil。Keil 公司推出的面向微控制器的集成开发环境 uVision 在广大单片机工程师中备受欢迎，因此，在收购次年，ARM 公司推出了采用 uVision 3.0 + ARM 编译器的 MDK 开发套件，该套件主要面向微控制器类应用的开发。为了满足用户在高端嵌入式微处理器上进行 Linux 和 Andorid 操作系统的开发，ARM 公司于 2010 年推出了全新的 DS-5 开发套件（按照 ARM 的说法，DS-5 的意思是 Deleopment Studio 第

5版的意思)。从ARM公司的产品战略上来看,其当前的主开发工具产品主要是面向高端应用的DS-5和面向控制器应用的MDK。

我们将在本节介绍这几代产品的主要特点。虽然ADS已经完成了它的历史使命,但对于初学者入门来说,该工具依然是一个不错的选择。

3.4.1 ADS 集成开发环境

ADS (ARM Developer Suit) 集成开发环境即 ARM 开发工具套件,如图3-27所示。

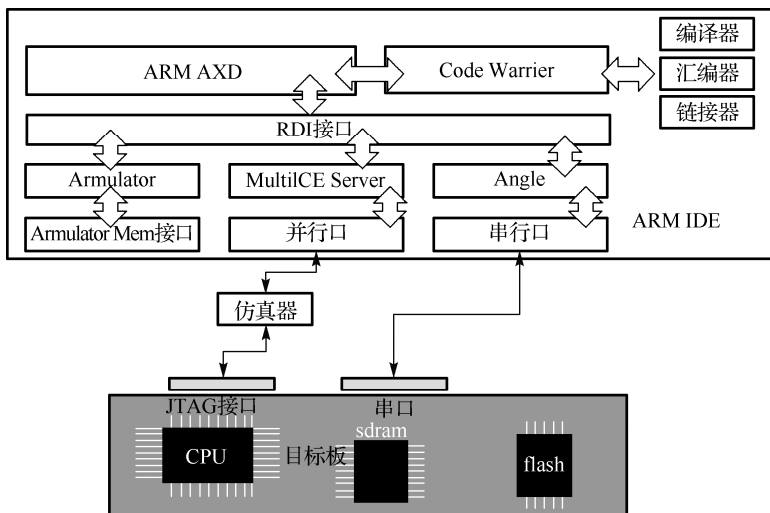


图 3-27 ADS 集成开发环境

ADS 集成开发环境中一个重要的工具就是 Code Warrior, Code Warrior 提供了项目管理的功能,并为编译链接提供一个环境,对应实现了编译器、汇编器和链接器的功能。除此之外,ADS 集成开发环境还提供了 AXD (ARM Extended Debugger) 工具,即 ARM 扩展调试器,在 AXD 中,定义了一个接口 RDI,ARM 支持的所有调试方式,诸如 Angle 调试、Armulator 模拟器调试、MultiICE 仿真器调试等调试接口都直接与 RDI 接口实现通信,不管使用的是哪种调试方法,由于直接与 AXD 通信的都是同一个 RDI 接口,所以得到的调试信息是相同的,包括断点的设置、单步程序执行的控制等,其调试机制都是相同的,这在很大程度上简化了学习 AXD 的困难。这里要特别提一下 MultiICE 仿真器,实际上,此时的 MultiICE 已经不再是仿真器了,它运行 MultiICE Server 程序,实现了 PC 协议与 JTAG 协议之间的协议转换功能,所以更确切地说,它是一个协议转换器。

3.4.2 DS-5 集成开发环境

自 2010 年 3 月 DS-5 第 1 版发布以来,很多熟悉 ADS 和 RVDS 的用户都慢慢转向使用 DS-5 这款支持所有 ARM 内核的开发工具。目前,最新的 DS-5 版本是 5.22。

1. DS-5 中的编译器

从历史渊源来说,DS-5 已经进化了三代,到它已经是第四代了。第一代是 ARM SDT,第二代是 ADS,第三代是 RVDS, RVDS 更新到 4.1 版本就停止了,之后便是 DS-5 了。在整个 4 代中,ARM 都是使用同一个编译器 ARM Compiler,虽然在每一代的叫法不同,比如 RVDS 的时候称为 RVCT,但其实是同一个编译器。与之前几代工具不同,DS-5 除使用最新版 ARM Compiler 5 外,还加了一款称

为 Linaro GNU GCC Compiler for Linux 的编译器，该编译器可以对 Linux 应用程序进行开发，这就使得 DS-5 可以覆盖 ARM 嵌入式开发的整个过程，保证了底层到应用层开发的连续性。

DS-5 专业版中的 ARM 编译器是唯一一个结合 ARM 处理器开发的商用编译器，它由 ARM 公司自己的编译器团队开发，专门用于为 ARM 架构提供最佳支持。该编译器的开发历经 20 年，被公认为是业界标准的 C 和 C++ 编译器，用于生成面向 ARM、Thumb、Thumb-2、VFP 和 NEON 指令集的应用程序。

高效的 ARM 编译器采用强大的优化技术，如循环展开、函数内联、常见函数识别、架构特定指令调度以及 NEON 自动向量化等。与 RVDS 4.0 编译器相比，ARM 编译器 v5.0 将其最高性能提高达 15%。通过 ARM 编译器先进的 NEON 自动向量化功能，可以从标准 C 和 C++ 源代码自动生成 ARM NEON SIMD 代码序列。向量化可以提高关键多媒体内核的速度，最高达到原来的 4 倍。

与其他编译器相比，ARM 编译器融入了一些可将应用程序的占用空间减少达 30% 的技术。ARM 编译器随附了一个可选 `microlib` C 库，该库的运行时库代码大小已减到最小，减小的幅度高达 50%。与 RVDS 4.0 编译器相比，ARM 编译器 5.0 将最佳代码大小减小多达 5%。与 Cortex-M 系列处理器结合使用时，`microlib` C 库可提供完全基于 C 语言的开发环境，无需恢复至汇编语言——即使对于中断服务例程也是如此，从而无需具体了解 ARM 架构。

无论是在 Cortex-A15 处理器上创建面向 NEON SIMD 的现代多媒体应用程序，还是重新构建一个面向 ARM7TDMI 的已有 10 年历史的旧库，都只需要一个 DS-5 专业版许可证。这是因为，ARM 编译器 5.0 支持所有 ARM 内核。此外，也可以对某个旧代码库进行较小更改并使用原始编译器来重新生成，因为 DS-5 ARM 编译器许可证与 ARM 编译器的所有先前版本向后兼容，包括 RealView 编译工具 (RVCT) 和 ARM 开发套件 (ADS)。

ARM 编译器堆栈保护功能可针对恶意返回地址修改提供保护，这种修改可能导致不需要的软件探查设备上的敏感数据或通过重定向控制流对其进行控制。ARM 编译器堆栈保护功能在易受攻击的缓冲区与堆栈帧返回地址之间插入随机生成的堆栈“canary”，这样就可以在设备数据遭到破坏之前检测到堆栈攻击。堆栈保护机制也可用于检测无意中的堆栈缓冲区覆盖，从而提高系统的故障检测水平。

ARM 编译器选择的默认库提供完整的 C/C++ 功能，包括 C++ 异常处理和 IEEE 754 浮点支持。该编译器可选择性地生成代码以使用软件浮点或任何 ARM 硬件浮点单元。不论使用哪种方法，该编译器均可生成符合 IEEE 754 标准的完整代码。这意味着应用程序将生成完全相同的数据，而不论采用哪种目标处理器，从而加快了从某一设备到其他设备的移植。

2. DS-5 中的调试器

DS-5 调试器兼具集成微控制器工具的方便性和高生产率以及针对 Linux 开源工具的强大功能和灵活性。该调试器基于 Eclipse 的 GUI 简化了管理不同目标连接类型的复杂工作，可以对裸机程序（含 U-Boot）、Linux Kernel、Android Kernel、Linux App 和 Android App 进行调试，能进行单步调试、运行、断点、修改变量等操作。在调试界面中，有线程列表、堆栈、调试命令、变量、断点、寄存器、函数、源代码、反汇编代码、寄存器和存储器等资源情况。

专业的端到端调试器：直观的单用户界面，可用于从启动加载程序到应用程序调试的所有软件层；由 ARMCC 和 GCC 编译器生成的调试代码；可为软件开发提供可靠性和实时响应；一个许可证可用于所有支持的 ARM 应用程序核心。

高级会话控制和系统视图：从单个调试器环境控制针对一个或多个目标的多个同时发生的调试会话；无缝支持运行和停止模式调试；完整的系统可见性，包括内存、CPU 寄存器、外围寄存器、帧缓冲区等。

非侵入性跟踪分析：具有源代码同步功能的 ETM 和 PTM 指令跟踪；STM 和 ITM 测量跟踪支持；用于跟踪日志分析的指令和函数视图；具有 DSTREAM 的最多 4GB 的片外跟踪缓冲区。

OS 感知调试器：Linux 和 Android；FreescaleMQX RTOS；Keil RTX CMSIS-RTOS 兼容。

可提高生产率的功能：自动化工作流，连接、下载和运行；与 GDB 相似的命令行控制台和脚本撰写功能；目标文件系统浏览器；设备数据库和可定制的闪存编程。

DSTREAM 高性能调试和跟踪：运行控制调试和跟踪单元支持所有 ARM 和 Cortex 处理器；通过 USB 2.0 和以太网接口，可以从主机 PC 进行直接和远程连接；支持 JTAG 和串行线调试目标接口；代码下载速度最高可达 2500KB/s；频率高达 60MHz 的 JTAG 时钟可在现有调试端口提供快速软件上载；通过 300MHz DDR 进行 16 位宽度的跟踪捕获（600 兆比特/秒/针）；灵活的跟踪时钟定位（相对于跟踪数据）；高达 4GB 的跟踪缓冲区可以对快速目标进行长时间跟踪；设备引入和测试实用工具；支持通过 JTAG 进行虚拟以太网链接；支持第三方 IP 和调试器。

1) 会话控制

调试会话配置：可以通过简单的对话框轻松地配置新的 DS-5 调试器连接。在这个对话框中，用户从支持的平台的数据库中选择其设备/开发板、定义调试会话的类型（裸机、Linux 应用程序或 Linux 内核和驱动程序）、加载符号文件以及设计目标环境变量等。在保存后，仅需双击即可随时重新调用会话配置，从而在每次调试交互操作时节省开发人员的宝贵时间。

调试控制视图：使用调试控制视图可以非常轻松地管理调试连接、展现上下文和控制程序执行。调试链接在此视图中列出，并且只需一次单击即可启动连接，而与连接类型（例如 JTAG 或 GDB）无关。在顶部栏中，开发人员可利用运行控制命令来执行程序，这具有很高的灵活性。在由于谨慎处理或由于断点命中而暂停执行时，进程、线程和调用堆栈信息将立即显示。最终，调试控制视图允许与一个或多个目标同时建立多个连接，并且在用户将重心放在某个连接时自动刷新所有其他调试器视图。

源代码和单步汇编：对于对低级别调试或优化感兴趣的开发人员，DS-5 调试器提供了一个反汇编视图，可用于自动或手动地对 ARM 或 Thumb 指令进行编码。通过与程序执行完全同步，该反汇编视图突出显示了与所选源代码行匹配的所有指令，并且还可以用于逐个步骤地执行机器指令以及源代码级别。从调试控制视图，可通过运行、暂停和单步执行/单步跳过/单步跳出图标对程序执行进行控制。

命令行接口和脚本撰写：DS-5 调试器实现自己的类似于 GDB 的命令行接口，扩展了 GNU 工具的功能，并且包含 JTAG/SWD 和 GDB 连接。DS-5 调试器命令提示符可从 DS-5 IDE 内访问，也可以通过控制台从外部访问，前者还具备自动完成功能和工具提示帮助以方便使用。所有调试器操作（CLI 或 GUI 驱动的）都显示在命令视图中并且记录在历史记录视图中，这允许用户只需通过选择文本并在脚本视图中拖放脚本，即可将任何命令序列转换为脚本。然后，用户可以随时调用已保存的脚本，或者在初始化时或命中断点时自动执行已保存的脚本

断点和数据观察点：断点和观察点已成为软件调试的重要一环，但 DS-5 调试器令的功能更强大且更易于使用。软件和硬件断点可以轻松地从源代码中进行设置，并且在断点视图中与观察点一起列出。此视图用于禁用、启用、删除或配置断点和观察点属性，以便确保执行在正确的上下文中停止。“断点属性”对话框允许开发人员设置停止条件表达式、忽略计数、运行保存的脚本和自动继续执行，甚至允许仅在特定的线程停止。同样，DS-5 调试器中的数据观察点也非常灵活，并且可在以特定方式访问内存位置时、表达式计算结果为 true 时或者在到达了忽略计数时触发。数据观察点从变量视图或内存视图中进行设置。

2) 跟踪

非侵入式跟踪：在调试软件时，在许多情况下，错误的负面影响显而易见，但导致出现此类错误的根本原因却远在程序执行的早期就已存在。DS-5 调试器所支持的 ARM CoreSight ETM 和 PTM 提供非侵入性程序跟踪，允许开发人员在出现错误时查看说明（以及关联的源代码）。它还使开发人员能够

调试对时间敏感的问题，而用传统的侵入性单步技术将很难发现这些问题。DS-5 调试器使用 DSTREAM 将跟踪数据捕获到其 4GB 缓冲区中。

灵活的跟踪显示：跟踪是用于调试和短期性能分析的宝贵工具。但是很少开发人员能够只通过长长的已执行机器指令列表做到这一点。在 DS-5 中十分重视以高级语言开发人员方便了解的方式来展示这些数据，例如将指令链接到相应的源代码、显示功能级别跟踪分析或者提供图形跟踪显示。

基于跟踪的分析：基于跟踪数据，DS-5 调试器还生成时间表图表，所含信息可帮助开发人员迅速理解其软件在目标上的执行方式以及哪些功能最占用 CPU。通过提供不同的缩放级别，该时间表可基于每个时间单位的指令数以其最高分辨率显示热点地图，并且提供按每组指令的典型延迟时间以不同颜色编码的每组指令显示。

3) 系统视图

寄存器视图：在 DS-5 调试器内核中，协处理器和外设寄存器全都在寄存器视图中列出（屏幕快照）。此层次结构树视图以逻辑方式将寄存器和位字段进行分组，使用户可以轻松导航其内容。所有寄存器和字段都进行标记并根据制造商的官方文档进行验证，因此用户不会由于在无穷无尽的数据表中搜索地址、位字段和可接受值而浪费宝贵时间。此外，开发人员可以随时冻结此视图，使所有内容不可更改以便与以后的快照进行比较。

内存、变量和表达式视图：有许多方法可以使用 DS-5 调试器查看和修改内存内容。内存视图是传统的以线性方式展示内存空间的视图。此视图对于低级别调试非常有用，这体现在：可对此视图进行配置以便以不同的格式（默认为十六进制）和宽度显示数据，接受表达式和符号作为起始地址，并且可用于在特定的内存地址设置数据观察点。功能强大的变量视图将提供所有局部和全局变量的内容、类型、大小和地址的逻辑和上下文可视性。与寄存器视图相似，此视图也以黄色突出显示在前一步骤/运行中已修改的变量的值。最后，表达式视图提供快捷的方式来计算常用表达式和变量的值。

屏幕视图和半主机：为简化负责板引入/低级别调试的开发人员的工作，DS-5 提供了屏幕缓冲区查看器和通过调试代理的 I/O 通道功能。该屏幕视图可在主机上生动地展现屏幕缓冲区的内容，允许开发人员无需连接任何硬件即可显示视频输出。同样，可以在其他外设及其驱动程序启动前使用半主机作为控制台的标准 I/O。

4) Linux 识别

上下文识别：DS-5 调试器可提供处理器内核、操作系统进程和线程以及调用堆栈的完整情况。调试控制视图在每个调试连接的基础上通过直观的层次分明的布局显示所有这些信息，使得开发人员可以非常轻松地标识当前上下文以及堆栈帧之间的关系。此外，还为基于 Linux 的系统调试自定义了 DS-5 断点，实现了特定于线程的条件断点以及内核模块中尚未插入的挂起断点。

远程系统浏览器：通过远程系统视图，可以方便快捷地访问在基于 Linux 的系统开发中常用的服务。DS-5 是在安全外壳连接（SSH）基础上建立的，它嵌入了远程文件系统浏览器。在该浏览器中，可以对文件和目录进行完全管理和直接编辑，并且可以轻松地在主机和目标之间转换，如同拖放操作一般简单。此外，在 IDE 中，可以直接从远程系统视图随时启动远程终端视图。

模块视图：模块视图是特定于操作系统的生产率提高工具，可加快涉及共享库和内核模块的调试活动的速度。此数据面板可用于展现和管理正调试的应用程序所使用的共享库以及自连接了调试器后安装的内核模块。此外，在调查其中一个或多个库或模块内所发生的情况变得重要时，开发人员可以轻松地使用模块视图加载其调试信号。

3. DS-5 DSTREAM 高性能调试和跟踪单元

DSTREAM 高性能调试和跟踪单元支持任何基于 ARM 处理器的硬件目标上的软件调试和优化。

DSTREAM 扩展了 ARM AVI 和 ARM RVT2 的功能,内存下载和跟踪捕获的速度更快,同时为物理调试和跟踪接口提供了更广的支持。借助于 DSTREAM,可以通过 JTAG 或串行线调试(SWD)将 DS-5 调试器连接到基于 ARM 的设备。DSTREAM 使用 FPGA 加速,在单核处理器和多核处理器设备上实现现代码的高速下载和快速步进。

DSTREAM 是一个用于开发和调试复杂 SoC 的全面解决方案,具有许多功能。例如,针对多种开发平台引入加速硬件,并具有用于第三方工具的开放式调试接口。跟踪是用于解决复杂的软件/硬件问题以及与时序相关问题的必不可少的工具。因为通过跟踪,可以对软件的执行情况进行后分析,而不需要借助任何软件或硬件工具。DSTREAM 的 4GB 跟踪缓冲区可以长时间进行高带宽跟踪,提供了有关软件在目标处理器上的执行情况的更多信息。DSTREAM 随附了功能强大的软件实用程序,以帮助进行 SoC 硬件验证。它还提供了用于第三方工具和自定义工具的接口。



图 3-28 DStream 高性能调试与跟踪单元

Dstream 的特点如下:

- ① 运行控制调试和跟踪单元支持所有 ARM 和 Cortex 处理器。
- ② 通过 USB 2.0 和以太网接口,可以从主机 PC 进行直接和远程连接。
- ③ 支持 JTAG 和串行线调试目标接口。
- ④ 代码下载速度最高可达 2500KB/s。
- ⑤ 频率高达 60MHz 的 JTAG 时钟可在现有调试端口提供快速软件上载。
- ⑥ 通过 300MHz DDR 进行 16 位宽度的跟踪捕获(600 兆比特/秒/针)。
- ⑦ 灵活的跟踪时钟定位(相对于跟踪数据)。
- ⑧ 高达 4GB 的跟踪缓冲区可以对快速目标进行长时间跟踪。
- ⑨ 设备引入和测试实用工具。
- ⑩ 在 JTAG 第三方 IP 和调试器支持的基础上还提供虚拟以太网链路支持。

4. DS-5 中的 CPU 核模拟器

早期版本的 DS-5 中集成了不同 ARM CPU 核的模拟器,ARM 将其称为实时模拟器模型(Real Time Simulation Model, RTSM)。从 DS-5 版本 5.15 开始,RTSM 被改名为固定虚拟平台(Fixed Virtual Platforms, FVP)。

通过 DS-5 中的 Cortex-A8 和四核 Cortex-A9 模拟器,无需硬件目标即可进行软件开发。RTSM 是 ARM 硬件平台的仿真模型,其中包括运行复杂操作系统和应用程序所需的处理器、内存控制器和外设。

1) ARM RTSM 的主要优点

- ① 支持 ARM 处理器软件的早期测试和调试。
- ② 在典型的桌面 PC 上,仿真速度超过 250MHz。
- ③ 高质量程序员视图模型。
- ④ 将使用主机 PC 上的资源仿真外设接口,包括 LCD 控制器、键盘、鼠标、触摸屏、UART 和以太网控制器(这一点比早期的 Armulator 方便多了)。
- ⑤ 使用 CADI 连接对裸机或 Linux 内核开发进行停止模式调试。
- ⑥ 使用 gdbserver 连接对 Linux 应用程序开发进行运行模式调试。
- ⑦ 启动 Linux 映像示例。

2) ARM 嵌入式 Linux 示例

DS-5 包括移植到 Cortex-A8 RTSM 的示例 ARM Linux 版本, 用于开始进行基于 ARM 的 Linux 软件开发。当从 DS-5 中启动时, 此模型自动引导 Linux 并进入就绪状态, 以便用户加载和调试应用程序。

5. DS-5 中的 IDE

DS-5 的所有功能都基于 Eclipse IDE, 这款 IDE 的特点是项目管理方便并且可移植性强, 完全支持 Windows 和 Linux 两大主流系统, 并可以自行加入第三方插件, 具有高度的灵活性。在这样的环境下, 使用 DS-5 可以轻松地进行裸机程序、U-boot、Linux 内核、Android 内核、Linux 应用程序、Android 应用程序、实时系统、安全应用等一系列开发调试。开发人员可以只学习一套 IDE 的使用就可以进行各种类型程序的开发。

1) Workbench 和 IDE 概述

DS-5 基于标准 Eclipse 开发环境, 提供一流的窗口管理、项目管理和 C/C++源代码编辑工具。用户可以将 DS-5 作为独立 Eclipse 进行安装或作为现有 Eclipse 环境的插件进行安装。

2) 源代码编辑

Eclipse IDE 功能齐全的 C/C++源代码编辑器可以帮助你更多时间用于编写代码, 减少更正语法错误的时间。

列出函数、变量和声明的大纲视图; 突出显示 C/C++源代码中的语法错误; 针对 C/C++和 ARM/Thumb/Thumb2 汇编的可配置语法颜色方案和代码格式; 完整的更改历史记录, 可以与常见的源代码控制系统(包括 CVS 和 SVN)集成。

3) 到目标的文件传输

DS-5 包括一个远程系统浏览器(RSE)视图, 可以将应用程序和库轻松传输到目标上的 Linux 文件系统。

提供到目标的 FTP 连接, 可以浏览其文件系统、创建新文件夹以及从主机拖放文件; 通过在 FTP 视图中双击相应的文件来打开目标文件系统上的文件。在 Eclipse 中编辑它们并将它们直接保存到目标的文件系统; 通过 Shell 和终端窗口可以在目标系统上运行 Linux 命令, 而无需显示器和键盘; 显示在目标上运行的进程列表。

4) 窗口管理

Eclipse 中的灵活窗口管理系统使得可以充分利用可视工作区。

支持多个源代码和调试器视图; 根据需要排列窗口: 浮动(分离)、停靠、选项卡或最小化到“快速视图”栏; 通过将分离的窗口拖放到其他显示器支持多屏幕设置。

6. DS-5 中的性能分析

ARM Streamline 性能分析器是 ARM DS-5 工具链的一部分, 它使软件开发人员能够充分利用基于 ARM 处理器的系统上的可用资源, 以创建高性能和高能效的产品。它配有直观的图形用户界面, 可显示从 CPU 和 GPU 性能计数器到源代码热点再到实际功耗等信息, 这样, 开发人员就可以方便地缓解性能瓶颈, 改进代码并行度, 延长电池寿命并增强用户体验。Streamline 以系统跟踪点、硬件和软件性能计数器、基于样本的分析和用户注视为基础, 提供了用于软件优化的功能强大而灵活的系统分析环境。

Streamline 可以对 Linux 和 Android 应用程序进行跟踪, 跟踪数据保存在电脑上, 保存容量不受芯片内存限制, 由电脑硬盘容量决定, 保证可以长时间进行数据跟踪。Streamline 由多种视图构成, 其中最明显的就是 Timeline 视图, 这里可以动态观察 CPU、GPU、RAM、BUS、Cache、线程和功耗状态。使用 Call Paths 等视图轻松分析应用程序瓶颈, 改善代码效率。

系统级时间表分析：时间表视图将多个数据源组合在一起，为软件开发人员提供了完整的高性能板。通过这种基于时间的图形可视化，可方便地观察系统性能指标随数据捕获时间的推移而发生的变化。在性能计数器中发现了即时热点或虚假变化之后，可以在图形中选择感兴趣的时间段以生成有重点的分析报告。

配置文件向下提取：Streamline 性能分析器支持基于时间和基于事件的采样（EBS），以便对本机 C/C++ 应用程序中的代码热点进行深入调查。通过此功能，开发人员能够根据统计数据给进程、线程、函数、源代码行和汇编指令分配处理器时间或硬件计数器，如高速缓存未命中次数和执行的 ARM NEON 指令次数。

SMP 系统的内核感知分析：由于线程同步较差和并发不佳等问题，很多时候达不到多核 SoC 的潜在性能增益。对于 SMP 平台，Streamline 具有基于每个内核的性能计数器图表和 X 射线可视化模式，此模式可以映射每个内核进程和线程活动，这样，开发人员就可以显示代码的分布情况。

集成 ARM Mail 的图形性能分析：复杂用户界面和游戏内容等图形密集的任务并不是仅在一个处理器中孤立运行的。因此，开发人员需要能够看到跨应用程序和图形处理器的性能。Streamline 性能分析器可向上链接到 Mail 驱动程序以提供有关 OpenGL ES1.1 和 OpenGL ES2.0 使用情况的广泛统计信息以及 300 多个软件和硬件性能计数器，并对帧缓冲区进行采样以获取新的高性能和高效内容。

用户注释：Streamline 性能分析器通过一个简单而功能强大的解决方案对调试和性能分析加以协调——代码注释。从按时间表跟踪计算机状态变化到将帧缓冲区内容与性能问题交叉关联，简单代码测量可将软件与性能分析链接在一起。为此，只需从用户或内核空间写入 Streamline 的内核模块（gator）驱动程序。

3.4.3 MDK 集成开发环境

Keil MDK-ARM 是一个完整的软开发工具包，用于基于 ARM 处理器的微控制器软件开发。它适用于基于 ARM Cortex-M 系列、ARM7、ARM9 和 Cortex-R4 处理器的嵌入式应用程序。它包含众多示例、项目模板和中间件库，具有 TCP/IP 协议栈、Flash 文件系统、USB 主机和设备协议栈、CAN 访问以及图形用户界面（GUI）解决方案。

Keil uVision IDE 特点如下：

- ① 完全支持 Cortex-M、Cortex-R4、ARM7 和 ARM9 设备。
- ② 行业领先的 ARM C/C++ 编译工具链。
- ③ uVision4IDE、调试器和模拟环境。
- ④ Keil RTX 确定性、占用空间小的实时操作系统（具有源代码）。
- ⑤ TCP/IP 网络套件提供多个协议和各种应用程序。
- ⑥ USB 设备和 USB 主机堆栈配备标准驱动程序类。
- ⑦ ULINKpro 支持对正在运行的应用程序进行即时分析并记录执行的每条 Cortex-M 指令。
- ⑧ 有关程序执行的完整代码覆盖率信息。
- ⑨ 执行性能分析器和性能分析器支持程序优化。
- ⑩ 大量示例项目可帮助快速熟悉 MDK-ARM 强大的内置功能。
- ⑪ 符合 CMSIS Cortex 微控制器软件接口标准。

1. ARM 编译工具

MDK 中集成的 armcc 编译器来自 ARM 公司自己的编译器团队，本质上 MDK 中所使用的编译器与 DS-5 中所集成的 ARM 编译器是同一个编译器。（当然，DS-5 为了支持 Linux 和 Andorid 操作系统，

还集成了 GCC 编译器。)关于 ARM 编译器的介绍,读者可以参阅 3.4.2 节中的内容。

值得一提的是与 MDK 编译器一起发布的软件包中还包括了 MicroLib C 库, MicroLib 是适用于以 C 语言编写的基于 ARM 的嵌入式应用程序的高度优化库。与 ARM 编译器工具链附带的标准 C 库相比, MicroLib 可提供显著的代码大小优势,而这一点对许多嵌入式系统来说至关重要。

MicroLib 与标准 C 之间的主要差异如下:

- ① MicroLib 是为深度嵌入式应用程序设计的。
- ② MicroLib 经过优化,使用的代码和数据内存比 ARM 标准库更少。
- ③ MicroLib 设计为独立于操作系统运行,但这不妨碍它与任何操作系统或 RTOS (如 Keil RTX) 一起使用。
- ④ MicroLib 不包含文件 I/O 或宽字符支持。
- ⑤ 由于 MicroLib 经过优化,大大缩减了代码大小,因此一些函数的执行速度将比 ARM 编译工具中提供的标准 C 库例程要快。
- ⑥ MicroLib 和 ARM 标准库均包含在 Keil MDK-ARM 中。

要在嵌入式应用程序中使用 MicroLib,请在 uVersion 中选中“MicroLib”复选框并编译应用程序。uVersion 可将程序与 MicroLib 相链接,并快速方便地减小程序大小。

2. 中间件库

与 MDK 一起发布的套件中还包括大量非常实用的软件中间件库,用户可以非常方便地基于这些现成的软件模块构建自己的软件项目。使用中间件的好处包括:

- ① 快速开发可靠并且功能强大的应用程序。RTX 内核及其源代码提供了创建和控制多线程、实时应用程序所需的所有资源,并可按照系统的确切要求进行量身定制。多数 Keil 中间件库都可以与 RTX 配合使用或单独使用。
- ② 按需使用。MDK 专业版包括 TCP/IP 网络协议栈、CAN 协议、USB 协议和 Flash 文件系统等中间件。使用这些现成的软件资源可以让开发人员专注于应用程序的开发,而不需要浪费时间重新实现这些底层函数。
- ③ 利用 Keil 的专长。所有 Keil 中间件库均由 ARM 和 Keil 工程师针对 ARM 平台进行了设计、测试和优化。这些库采用模块化设计并使用简单的 API。开发人员可以非常方便地调用这些库。
- ④ 源代码。一经请求,可以源代码形式提供所有中间件组件。可以重新构建库或扩展其功能集,当需要源代码进行产品认证时也非常有用。
- ⑤ MDK 专业版中间件库是免版税提供的。虽然可以自己开发类似的功能,但这种方法很难降低成本,多数情况下会导致非预期的延迟,尤其是当工程师在不熟悉领域工作时。MDK 专业版使得能够降低成本,缩短开发时间。

MDK 专业版所包含的中间件库主要有以下内容。

1) RTX 实时操作系统

虽然不使用 RTOS 也能创建实时程序(通过在超级循环中执行一个或多个函数),但是 Keil RTX 这样的 RTOS 可以解决许多调度、维护和计时问题。Keil RTX 是免版税的确定性实时操作系统,适用于 ARM 和 Cortex-M 设备。使用该系统可以创建同时执行多个功能的程序,并有助于创建结构更好且更易维护的应用程序,如图 3-29 所示。RTX 的几大特点如下:

- ① 带有源代码的免版税、确定性的 RTOS。
- ② 灵活的调度:循环、抢先和协作。

- ③ 低中断延迟：以执行高速实时操作。
- ④ 空间占用小，以适用于资源受限的系统。
- ⑤ 不限数量的任务，每个任务都具有 254 个优先级。
- ⑥ 不限数量的邮箱、信号、互斥函数和计时器。
- ⑦ 支持多线程和线程安全运算。
- ⑧ MDK-ARM 中的内核识别调试支持。
- ⑨ 用户可以使用 uVersion 配置向导对 RTX 进行配置。

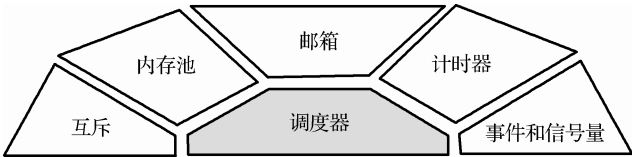


图 3-29 RTX 实时操作系统

2) TCP/IP 网络套件

该中间件是完整的 TCP/IP 网络套件，其专门针对小型、基于 ARM 和 Cortex-M 处理器的微控制器进行编写。它进行了高度优化，具有代码量小并且提供最佳性能的特点。该中间件的基本架构如图 3-30 所示，其主要特点包括：

- ① 完全支持 TCP/IP 和 UDP 协议。
- ② 以太网和串行（PPP 和 SLIP）物理层支持。
- ③ 包括通用网络应用程序——无需额外付费。
- ④ 灵活的内存占用空间——大小取决于使用的协议和应用程序。
- ⑤ 高速实现——已针对基于微处理器的系统进行优化。
- ⑥ 独立操作——可在没有 RTOS 的情况下使用。
- ⑦ 新功能：
 - 完全支持基于 Cortex-M1 的设备。
 - 提供了新的 FTP 服务器和 SNMP 代理实现。
 - Ajax——异步 JavaScript 和 XML。
 - SOAP——简单对象访问协议。
- ⑧ 广泛的调试支持。

该网络协议栈套件使用标准 C 语言进行编写，并使用 MDK 专业版中提供的 ARM 编译工具链进行编译。使用 TCP/IP 网络套件的应用程序只需包含一个专用头文件并使用 MDK 专业版中包含的库链接程序。

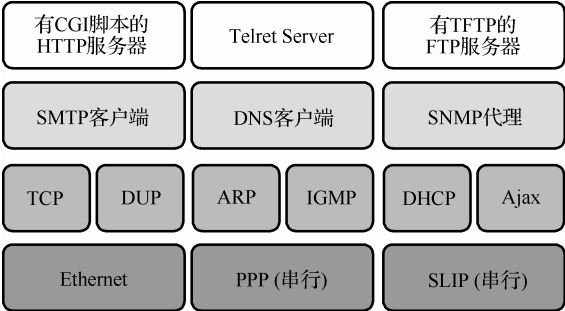


图 3-30 MDK 中的 TCP/IP 网络套件

3) Flash 文件系统

许多基于 ARM 的微控制器对标准文件系统具有实际要求。使用 Flash 文件系统，可以在嵌入式应用中实现新功能，例如数据记录、在待机模式期间存储程序状态或存储固件升级。

MDK 专业版包括 Flash 文件系统，此文件系统允许嵌入式应用程序在 ROM、RAM、Flash ROM 和 SD/MMC/SDHC 内存卡等标准存储设备中创建、保存、读取和修改文件。该中间件基本框架如图 3-31 所示，主要特点包括：

- ① 标准 ANSI C 文件 I/O 应用程序接口。
- ② NOR 和 NAND Flash 支持。
- ③ RAM、ROM 和 SD/MMC/SDHC 内存卡。
- ④ Windows 兼容 FAT12、FAT16 和 FAT32 格式，包括子文件夹和长文件名。
- ⑤ SD/MMC 卡文件高速缓存提供高速读/写访问。
- ⑥ 可重入和线程安全运算。
- ⑦ 同时访问多个存储设备。
- ⑧ 可以使用 uVersion 配置向导进行设置。
- ⑨ 为众多基于 ARM 的设备提供片上 Flash 和外部内存接口支持。



图 3-31 Flash 文件系统

4) CAN 驱动程序

MDK 专业版包括通用 CAN 接口层，它为所有支持的微控制器提供标准的编程 API。它为实现 CAN 网络提供了一种既快速又简单的方式，并提供了代码可移植性，便于将代码迁移到其他微控制器，如图 3-32 所示。

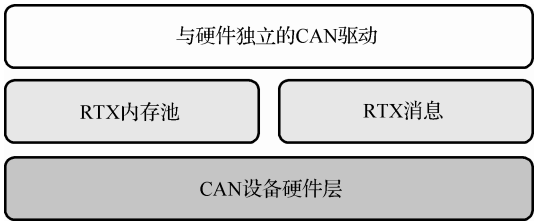


图 3-32 CAN 驱动程序

RTX 函数： CAN 接口使用 RTX Kernel 函数为邮箱管理和内存分配运行中断服务例程。它对所有 CAN 消息使用一个内存池，并使用两个邮箱数组。

通用软件层： 通用软件层允许用户在不同目标之间应用相同接口，轻松地从一個目标切换到另一个目标而不必更改主应用程序代码。**CAN 接口 API 支持：** 初始化/启动 CAN 通信；定义用于 RX/TX 的 CAN 消息对象；发送、请求和接收 CAN 消息。

硬件独立层： CAN 硬件层允许通用软件接口在很多不同的目标上工作，每个目标有自己的硬件层实现。硬件层支持以下设备：Atmel SAM7 和 SAM9；NXP LPC17xx、LPC21xx、LPC23xx、LPC24xx

和 LPC29xx; STMicroelectronics STR7、STR9 和 STM32; Texas Instruments Stellaris 系列。

CAN Primer: CAN 广泛用于汽车和卡车,但也用于其他领域。有很多应用程序曾可用于 CAN,不过,也可以很容易地开发你自己的协议以适合并简化你的需要。

5) USB 设备接口

MDK 专业版为嵌入式系统提供 USB 设备端和 USB 主机端支持。USB 设备接口使用所有 Windows PC 中都提供的标准设备驱动程序类,不需要开发 Windows 主机驱动程序。USB 设备接口使用通用软件层,该软件层使用 RTX 内核功能,如图 3-33 所示。

主要特点包括:

- ① USB 硬件层和事件处理程序(特定于硬件)。
 - ② 支持 USB 1.1 和 2.0 的通用 USB 内核。包括低速(1.5Mb/s)、全速(12Mb/s)和高速(480Mb/s)。
 - ③ 通用 USB 设备类支持人机接口设备(HID)、容量存储类(MSC)、音频设备(ADC)、通信设备(CDC)和复合设备。
 - ④ 与其他 MDK 专业版组件集成在一起。
- MSC 使用 Flash 文件系统来支持 SD/MMC 卡存储。
 - 使用 RTX 实时内核。

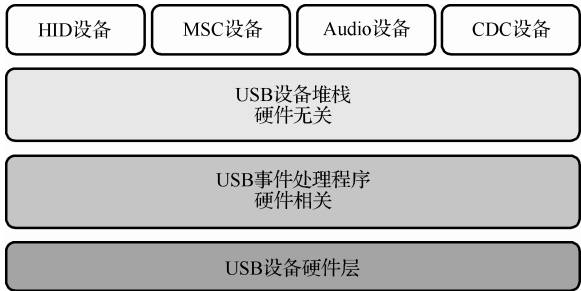


图 3-33 USB 设备接口

6) USB 主机接口

USB 主机库是支持 USB MSC(大容量存储类)和 HID(人机接口设备)类的嵌入式 USB 堆栈。它可以使用尽量少的内存提供高性能。库中包含的示例演示如何将 USB 大容量存储设备和 Flash 文件系统一起使用以及如何使用外部 HID 输入设备,如图 3-34 所示。

主要特点包括:

- ① 抽象层允许将一个标准 API 用于不同的 USB 主机控制器。
 - ② USB 主机控制器支持:
 - 通用开放主机控制器接口(OHCI)。
 - 针对 STM32F105/7(连接线)设备的自定义主机控制器驱动程序。
 - USB1.1 低速(1.5Mb/s)和全速(12Mb/s)。
 - ③ 通用 USB 设备类支持:
 - 人机接口设备(HID)。
 - 大容量存储类(MSC)。
 - ④ 高速(670KB/s)和小代码大小(~6KB)。
 - ⑤ 与其他 MDK 专业版组件集成在一起。
- MSC 使用 Flash 文件系统来支持 USB 闪存驱动器和 SD/SDHC/MMC 存储卡设备。
 - USB 主机与 RTX 实时操作系统一起使用。

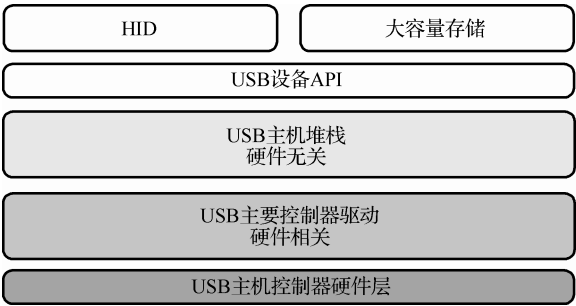


图 3-34 USB 主机接口

3. 实时跟踪和分析

MDK-ARM 使用 ULINK 适配器,在基于 ARM Cortex-M 系列处理器的设备中使用 ARM CoreSight 技术提供高级实时调试、跟踪和分析功能。

（1）数据和事件跟踪。

所有 Cortex-M3 和 Cortex-M4 设备都提供数据和事件跟踪。MDK-ARM 提供很多方法在系统运行时分析此类信息。

- ① 跟踪窗口：通过捕获时间戳、PC 样本、异常和读/写访问显示程序流。
- ② 调试查看器：在终端窗口中显示仪器跟踪（ITM）的 printf 风格的输出。
- ③ 异常窗口：显示有关程序异常和中断的统计信息。
- ④ 事件计数器：显示提供性能指示的特定事件计数器的实时值。
- ⑤ 逻辑分析器：以图形方式显示捕获的数据跟踪中的变量更改。
- ⑥ RTX 事件查看器：按时间段显示 RTX 内核任务切换事件和统计信息。

（2）指令跟踪。

使用 ETM 的 Cortex-M 系列处理器提供指令跟踪。Keil ULINKpro 流指令跟踪直接传入 PC，开发人员可以调试历史序列、执行分析、性能优化和代码覆盖率分析。

代码覆盖：几乎不受限制的跟踪信息流能够使 MDK 提供完整的程序代码覆盖率。代码覆盖率标识每个已执行的指令，从而确保对程序进行彻底测试。这是对完整的软件验证和认证的基本要求。代码覆盖率标识已执行的代码，可有助于确保彻底测试用户开发的应用程序。

性能分析器：性能分析器记录并显示所选函数和程序块的执行时间。条形图显示程序中每个部分花费的时间。可以使用由性能分析器收集的信息来确定应用程序中的执行热点（Hot Spot）。然后集中精力优化这部分代码，以使程序的总体性能得到提升。

逻辑分析器：逻辑分析器以图形方式显示随时间变化的信号和程序变量。使用逻辑分析器对话框中的设置按钮或通过拖动符号窗口中的符号可以轻松配置逻辑分析器记录的信号。使用光标键可对齐到信号变化。悬停鼠标指针可显示增量信息。逻辑分析器甚至可以转到信号变化时执行的指令或语句。

3.5 嵌入式软件的执行镜像与启动过程

通常情况下，编译器和链接器输出的文件〔注意这里的文件是调试主机（也就是 PC）上的文件格式〕是无法直接在嵌入式目标系统上直接运行的。这主要是因为两个原因：第一，链接器输出的文件往往要遵循一定的格式，不同的链接器输出格式可能各不相同，比如 ARM 链接器输出的格式采用

了标准的 ELF 格式。请注意, 这些输出文件并不是可以在嵌入式目标系统上直接执行的指令流, 而是包含了大量的调试信息和装载信息。前者用于为调试器软件提供相应的信息, 比如符号表 (也就是程序中所有的函数以及全局变量的名字以及所对应的物理地址); 后者则是为加载该程序的加载器 (Loader) 提供了加载信息, 比如可执行镜像应该被加载到目标系统的哪个地址, 如何初始化有初值的全局变量和没有初值的全局变量等。第二, 正如我们前面介绍的, 这个输出文件是存放在调试主机的, 必须通过一定的方式将其下载到目标系统, 并通过加载器将可执行镜像加载到特定位置后才可以运行。

关于上面所说的第二点, 情况比较复杂, 我们将分以下两种情况进行讨论。对于有文件系统和动态加载的嵌入式操作系统而言, 开发人员只需要将链接器输出的文件通过一定的手段将该文件写入目标系统的文件系统即可, 这种写入手段可以是 FTP 或者是网络文件系统 (Linux 操作系统都支持这些)。当需要运行这些软件时, 只需要调用操作系统提供的相关系统调用即可, 嵌入式操作系统将负责将该文件加载到目标系统的主存储器, 完成相应的初始化, 并最终将控制权交给该应用程序。操作系统还将负责用户应用程序执行退出后的后续处理, 比如释放所申请的相关资源等。这个过程与我们在 PC 上执行一个应用的过程是一样的。

对于没有操作系统, 或者操作系统不提供动态加载 (很多轻量级的嵌入式操作系统都不提供该功能) 的系统而言, 一般是将所开发的应用程序与操作系统合并在一个项目中进行编译和链接。也就是说, 在这种情况下, 操作系统和应用程序被链接器合并为一个统一的可执行镜像, 所开发的应用程序是作为一个操作系统的内部函数进行调用的。问题的复杂性在于不管是第一种情况 (操作系统加载用户应用程序文件) 还是第二种情况 (操作系统直接调用应用程序函数), 操作系统本身的可执行镜像该如何加载呢?

对于像 Linux 这样复杂的操作系统, 系统设计人员往往需要将整个系统的软件分为 3 部分, 首先是 Boot Loader, 其次是内核镜像, 再次是文件系统。其中 Boot Loader 和内核镜像都是独立的可执行镜像, 也就是说它们一般是分在不同的项目中独立编译和链接的。Boot Loader 的主要作用就是启动系统, 完成必要的初始化 (比如启动系统的锁相环, 以使系统的主频提升到正常工作的频率; 又比如初始化 SDRAM 存储器, 为 SDRAM 控制器设置相应的时序参数; 初始化 nand Flash 存储器; 等等), 在完成初始化后, Boot Loader 将根据预设的信息将保存在非易失存储器上特定位置的操作系统镜像加载到 SDRAM 特定的地址 (该特定地址应该在链接内核时指定) 并将控制权交给内核。内核接管系统后, 将进一步对整个系统进行初始化, 并挂接文件系统。自此, 内核就可以开始加载存放在文件系统库文件和应用程序了。

如果操作系统比较简单, 系统一般不需要单独的 Boot Loader 去加载操作系统, 因为系统启动后, 在完成必要的初始化之后 (该初始化代码也属于操作系统的一部分, 并且链接在一个执行镜像中) 直接跳转到操作系统的代码。对于这种情况, 首先要将完整的操作系统镜像烧写在目标系统的启动存储器中 (启动存储器一定是非易失存储器, 因为系统刚上电时 RAM 内容是随机值)。烧写在启动存储器的镜像是纯二进制可执行镜像, 而不是链接器输出的文件, 用户一般可以通过编译器提供商所提供的工具将链接输出文件转化为纯二进制镜像。然后通过相应的烧写工具将该镜像写入非易失存储器 (通常是 Nor Flash 或者是 Nand Flash)。

本节将以 ARM 处理器以及 ARM 工具链为例, 介绍嵌入式系统的可执行镜像与启动过程。

3.5.1 ARM 链接器的输出文件的加载视图与执行视图

正如 3.1 节所述, ARM 汇编器在将用户所编写的汇编文件汇编为目标文件时会将用户的程序划分为 3 个段 (Section), 即 RO 段、RW 段和 ZI 段^①。RO 是 Read Only 的简称, 汇编器将程序的可执行

^① 新版本的 ARM 汇编器和链接器增加了一个新的段, 称为 XO 段, 意为 Execute-Only。

指令流都合并到该段；RW 是 Read and Write 的简称，汇编器将把所有具有初值的全局变量合并到该段；ZI 段是 Zero Initialized 的简称，汇编器将把所有没有初值的全局变量合并到 ZI 段。之所以要区分 RW 段和 ZI 段，是因为用户的程序中所声明的全局变量有些设置了非零的初始值，而有些则没有设置初值，因此在该程序被装载到存储器时，除了为这些全局变量分配所需要的存储空间外，还必须将非零初值的全局变量设置初始值，而对于没有指定初值的变量则需要将其初始化为零。汇编器和链接器通过将这两类不同类型的全局变量划分到不同的端，以进行不同的初始化。

简单地说，链接器的作用就是将不同目标文件中所划分好的程序段进行重新打包，所有的 RO 段合并为一个大的 RO 段，所有的 RW 段合并为一个大的 RW 段，所有的 ZI 段合并为一个大的 ZI 段。在合并过程中有两个步骤是非常重要的：第一是如何确定合并后的各个大段从什么基址开始排列；第二是由于存放在各自目标文件中的程序段是独立编址的，而链接合并后需要统一编址，因此原来独立编址的段在合并后需要由链接器根据重新分配的基址对原来段内的所有地址进行重新定位，这个过程也被称为重定位（Re-allocation）。经过链接后的输出文件按照一定的格式（ARM 链接器遵循 ELF 格式）进行组织。我们可以称这个文件为可装载的镜像（但还未被装载），而不是可执行的镜像，因为该输出文件中的指令流和数据还没有被存放到由链接器指定（开发人员通过链接器选项来指定这些地址）的存储器中。

链接器输出的文件可以被加载到目标系统的指定地址（通常在链接时指定），这个加载过程可以由调试器软件（Debugger）、Boot Loader、操作系统中的应用程序加载器（Loader）或者烧写软件完成。我们将分别对这几种情况进行讨论。运行于 PC 端的调试器读取链接器输出的文件，并按照文件中指定的地址，通过 JTAG 接口和芯片内置的 ICE 电路将文件中的相应段写入目标系统的存储器^①（通常只能是目标系统的 RAM，而不能是非易失存储器）。对于需要将可装载镜像“装载”到非易失存储器的情况，通常必须借助专门的烧写软件。该烧写软件一般分为主机端软件和目标端软件，主机端软件首先通过目标端的 JTAG 接口将目标端软件下载到目标系统的 RAM 存储器，然后主机端软件再将需要烧写的相应段（RO 段和 RW 段）数据传输到目标系统的 RAM 中，待数据传输完后，主机端会给预先传入的目标段软件下发烧写命令，该命令包含需要写入的非易失存储器地址，由于目标端软件内置了非易失存储器的驱动，它能将缓存在 RAM 中的 RO 段数据和 RW 段数据烧写到指定的非易失存储器，如图 3-35 所示。通常 Boot Loader 和操作系统镜像都是由烧写软件在产品发布前首先由厂商烧写到目标系统的非易失存储器的。正如前文所述，Boot Loader 的主要功能就是加载操作系统，Boot Loader 在启动后将接管整个系统，在完成必要的初始化后（系统时钟、SDRAM 等），它将按照预先设置的配置信息，从指定的非易失存储器地址将存放在那里的操作系统镜像（该镜像也是采用烧写工具预先写入的）加载到 SDRAM 的指定地址，并将控制器交给操作系统镜像。而对于操作系统而言，可以通过专门的 Loader（有些时候也被称为 Lanucher）负责将用户应用程序的可执行文件（链接器输出文件）转载到 RAM 中。

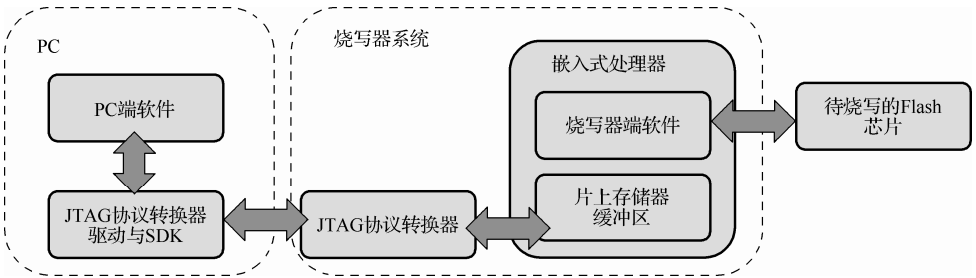


图 3-35 Flash 烧写器的软件框架

① 读者可以参阅本章 3.3.3 节了解通过 ICE 访问存储器的方法。

在完成镜像的加载后，链接器输出的程序段 RO 和数据段 RW 已经被存放在指定的位置，ARM 工具链称此时的状态为加载视图（Load View）。但加载视图还不是可以执行的状态，为了准备执行，链接器在用户开发的代码中还插入了一些初始化代码，在用户代码开始执行之前，某些程序段（RO 段或者 RW 段、ZI 段）可能会被搬移到新的存储器地址（具体如何搬运取决于用户链接时指定的参数）并完成初始化，比如 ZI 段通常在 Load View 时并不存在，链接器插入的初始化代码将在 RAM 区开设一块存储区并以零初始化这块区域。在完成该初始化后，就可以开始执行用户的代码了，此时 ARM 工具链称此时的状态为执行视图（Execute View）。

3.5.2 基于 ROM 的程序执行

一般来说，在低端产品中，为了节省 RAM，通常把程序放在 ROM（现在一般都是 Nor Flash 存储器）中执行，而把数据（如全局变量、动态申请的存储器和堆栈等）存放在 RAM 中。ROM 的值在一上电时就是确定的，而 RAM 的值是不确定的，所以一开始只能将 RO（read only）和 RW（read and write）段烧录（其实也就是加载到）到 ROM 中，在这个过程中，忽略 ZI（zero initial）段。但是因为 ROM 中的内容需要先擦除后才能进行写操作，如果 RW 放在 ROM 中，就会导致不可写，于是在系统上电后 RO 运行时，要将 RW 复制到 RAM 中，RW 在 RAM 中的地址由链接器决定，然后程序将属于 ZI 的地址段全部初始化为零，剩下的 RAM 地址作为动态内存分配和堆栈，而 RO 段就保留在 ROM 中运行，如图 3-36 所示。这样做的优点是节省了 RAM 空间，但也有缺点，通常来讲，ROM 的读取速度比 RAM 慢，Flash 一般要 60~70ns 才能完成一次数据的通信，对于主频为 100M 的处理器而言，取一条指令需要至少 6 个时钟周期的等待，这是比较耗时的。因此，对于大多数不是特别成本敏感的系统而言，通常采用下一种运行模式。

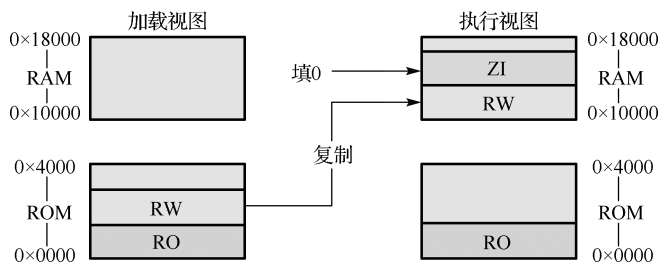


图 3-36 基于 ROM 程序执行的内存映射

3.5.3 基于 RAM 的程序执行

程序在 RAM 中执行时，一开始 RO 段和 RW 段都存放（加载）在 ROM 中，之后将 RO、RW 和 ZI 段复制到 RAM 中，程序就开始在 RAM 中运行，如图 3-37 所示。但无论采取哪种运行方式，都要通知链接器，通过程序员编写的 scatter 文件负责将 RO 段和 RW 段存放在对应的 RAM 地址中。

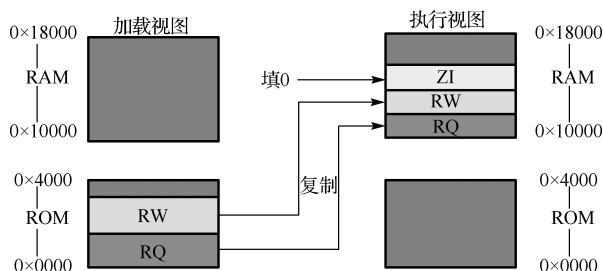


图 3-37 基于 RAM 程序执行的内存映射

3.5.4 ROM/RAM 重映射

系统启动都必须从 ROM 开始，不可能从 RAM 启动，因为系统上电时 RAM 的内容是不确定的。由于 ARM 的中断向量表一般放在零地址处，一旦复位，系统从零地址开始取第一条指令。但是访问 ROM 的速度较慢，而且 ROM 的中断向量表在执行过程中是不能修改的。为解决这个矛盾，通常需要 Remap（重映射）功能，系统启动后，ROM 区有两个地址：一个是零地址，一个是 ROM 片选所配置的地址。系统上电后，两个基址访问的地址是同一个物理地址，程序运行一段时间后通过配置存储控制器的 Remap 寄存器，零地址可以被重新映射到 RAM 区，而 ROM 区就只拥有片选信号所配置的地址了（此时 RAM 区拥有两个基址，也就是零地址和原来 RAM 片选所配置的基址）。在零地址重映射后可以在 RAM 中重建一个新的中断向量表，如图 3-38 所示。关于重映射机制，读者可以参阅本书的 2.2.2 节。

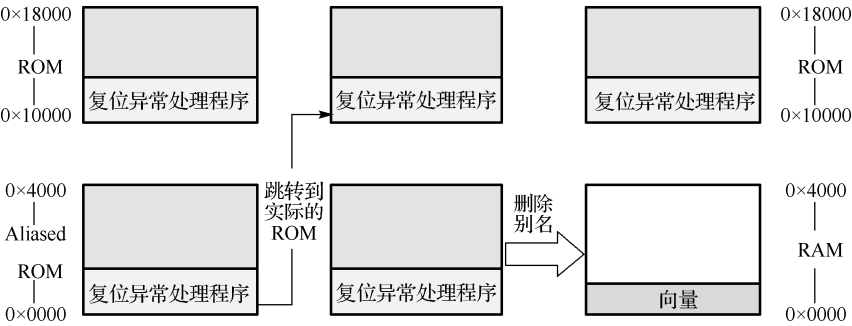


图 3-38 ROM/RAM 重映射

思考题

- 1. 分析利用 ADS 开发环境开发基于 ARM 的程序时，printf 功能是怎么实现的？
- 2. JTAG 中定义了 5 个不同的信号，其中一个比较明显的特点是 JTAG 标准采用了同步串行的数据传输方式，显然这比较低效，为什么不设计成为并行的数据传输方式呢？
- 3. 为什么 ARM 编译器输出的段只包括 RO、RW 和 ZI？程序中的临时变量保存在哪个段？

扩展阅读

[1] John R.Levine, Linkers and loaders[M], Morgan Kaufmann, 1999.

[2] Butko, A.; Garibotti, R.; Ost, L.; Sassatelli, G., "Accuracy evaluation of GEM5 simulator system," *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on* , vol., no., pp.1,7, 9-11 July, 2012.

[3] Binkert N, Beckmann B, Black G, et al. The gem5 simulator[J]. ACM SIGARCH Computer Architecture News, 2011, 39(2): 1-7.

[4] 吕彩霞. JTAG 的设计与研究[硕士学位论文]. 北京交通大学, 2006.

[5] 马俊, 雷航. 基于 SOPC 的通用型 JTAG 调试器的设计. 单片机与嵌入式系统应用, 2008, 7: 008.

[6] 韩可, 邓中亮, 施乐宁等. 针对 JTAG 调试的 RTL 验证环境设计原理. 电子测量技术, 2008, 31(1): 72-76.

- [7] 魏勇. 嵌入式交叉调试技术的研究与实现[硕士学位论文].电子科技大学, 2005.
- [8] 熊建. 基于 SOPC 的 ICE 调试器设计与实现技术研究[硕士学位论文]. 电子科技大学, 2005.
- [9] Jeppesen III J H, Clair-Hong K S S. JTAG interface system for communicating with compliant and non-compliant JTAG devices: U.S. Patent 5,708,773[P]. 1998-1-13.
- [10] Battaline R P, Robinson J R, Welbon E H, et al. Performance monitoring through JTAG 1149.1 interface: U.S. Patent 5,768,152[P]. 1998-6-16.
- [11] Whetsel L. A Standard Test Bus and Boundary Scan Architecture. Texas Instruments Technical Journal, 1988, 5(4): 48-59.
- [12] Bleeker H, van den Eijnden P, de Jong F. The Boundary-Scan Test Standard[M].Boundary-Scan Test. Springer US, 1993: 19-50.
- [13] Arm Corporation.ULINK2 Technical Specifacations [EB/OL]. <http://www.keil.com/ulink2/specs.asp>, 2010.
- [14] Arm Corporation.Multi-ICE User Guide[EB/OL]. <http://ecos.sourceware.org/multi-ice.html> , 2012.
- [15] Segger Corp.J-Link Performance Comparison[EB/OL]. <http://www.segger.com/performance-comparison.c-om>, 2013.
- [16] Roberto Mijat,Arm Corporation.Better Trace for Better Software-Introducing the new ARM CoreSight System Trace Macrocell and Trace Memory Controller [Whitepaper], 2010.

第 4 章 SoC 中的 CPU 内核

4.1 CPU 的基本概念

4.1.1 CPU 的发展

自从第一台通用电子计算机问世以来，计算机技术在大约 65 年间取得了令人难以置信的发展。今天，花费不到 500 美元购买的一台便携计算机，在性能、主存储器和磁盘存储方面都要优于 1985 年花费 100 万美元购买的计算机。这种快速发展既得益于计算机生产技术（尤其是微电子制造技术）的发展，也得益于计算机体系架构设计的创新。

制造技术的进步一直都相当稳定（自从 1965 年摩尔定律提出以来，微电子制造技术基本是遵循着单芯片集成度每 18 个月翻一番的速度向前发展的），而计算机体系结构的改进对这一快速发展的贡献就远没有那么稳定了。在电子计算机前 25 年的发展中，这两种力量都做出了巨大贡献。20 世纪 70 年代后期出现了微处理器。依靠集成电路技术的进步，微处理器也使计算机性能进入快速发展期——每年大约增长 25%。

这种高增长速度，再加上微处理器大量生产带来的成本优势，提高了微处理器业务在计算机行业内所占的份额。另外，计算机市场的两个重大变化也使新体系结构更容易在商业上获得成功。第一个重大变化是人们几乎不再使用汇编语言进行编程，从而降低了对目标代码兼容性的要求。第二个重大变化是出现了独立于厂商的标准化操作系统（比如 UNIX 和 Linux），降低了引入新体系结构的成本和风险。

正是由于这些变化，人们才有可能在 20 世纪 80 年代早期成功地开发了一组指令更为简单的新体系结构——RISC（精简指令集计算机）体系结构。设计人员在设计 RISC 计算机时，将主要精力投注在两种关键的性能技术上，即指令级并行（ILP, Instruction Level Parallel）的开发（最初是通过流水线，后来是通过多指令发射）和高速缓存（Cache）的使用（最初采用一些很简单的形式，后来使用了更为复杂的组织与优化方式）。

基于 RISC 架构的计算机抬高了性能指标，过去的体系结构要么快速跟上，要么就被淘汰。DEC 的 Vax 未能跟上时代的脚步，所以被采用 RISC 体系结构的 Alpha 处理器替代。Intel 则接受了挑战，在保证指令集兼容的前提下，80386 以后的处理器通过在内部将 80x86 指令转换为类似于 RISC 的指令，并大量采用 RISC 架构的相关技术（比如超深流水线、乱序执行、多指令发射等）。20 世纪 90 年代后期，单芯片所能集成的晶体管数目飞速增长，使得这种外壳保留 x86 指令集，而内部采用 RISC 技术的体系结构所造成的额外硬件开销和能耗开销对于桌面级和服务级应用而言可以被接受。然而，对于移动设备类的应用，比如在手机中，这种 x86 转换开销所带来的功耗与硅面积成本的上升是无法承受的，也正因为如此，促使一种 RISC 体系结构逐渐成为主流，这就是目前在移动应用领域独领风骚的 ARM 架构。

体系结构与组织方式的发展一起促成了计算机性能以超过 50% 的年增长率持续增长 17 年（1986 年~2003 年），这一速率在计算机行业内是空前的。

20 世纪的这一飞速发展共有四重效果。

第一，它显著增强了可供计算机用户使用的功能。对许多应用来说，当今性能最高的微处理器不到 10 年前的超级计算机还要优秀。

第二，性价比的这种大幅提高导致了新型计算机的出现。20 世纪 80 年代出现了个人计算机和工

作站，正是因为有微处理器可供使用。在刚刚过去的 10 年里，人们见证了智能手机和平板电脑的崛起，许多人把它们作为自己的主要计算平台，代替了个人计算机。这些移动客户端设备越来越多地通过因特网来访问包含数万个服务器的数据中心，这些数据中心的设计使它们对于用户而言看起来就像单个巨型计算机一样。（传说中的云计算？）

第三，根据摩尔定律的预测，半导体制造业的持续发展已经使基于微处理器的计算机在整个计算机设计领域中占据了主导地位。传统上使用现成逻辑电路或门阵列制造的小型机已经被使用微处理器制造的服务器所取代，甚至大型计算机和高性能的超级计算机也都是由微处理器组合而成。

这一硬件复兴还有第四个影响，那就是对软件开发的影响。自 1978 年以来，硬件性能提高了 25 000 倍，从而允许今天的程序员以性能换取生产效率。今天，绝大多数的编程是使用诸如 Java 和 C#之类的托管编程语言来完成的，代替了以提高性能为目的的 C 语言和 C++语言。此外，Python 和 Ruby 之类的脚本语言（它们的生产效率可能更高），连同 Ruby Rails 之类的编程框架，也正在日益普及。为了保持生产效率并尝试缩小性能差距，采用即时(Just-In-Time)编译器和跟踪编译(Trace-based Compiling)的解释器正在取代过去的传统编译器和链接器。软件部署也在发生变化，因特网上使用的“软件即服务”(SaaS, Software as a Service)取代了必须在本地计算机上安装和运行的盒装光盘套件(shink-wrapped)软件。另一方面，应用程序的本质也在发生变化。语言、音效、图像、视频正在变得愈加重要，对于用户频繁交互的移动智能终端类应用而言，响应时间对于提供良好的用户体验非常关键，而且我们相信，在可预见的未来这种重要性愈加重要。

令人遗憾的是，“万物有始必有终”，这一长达 17 年的硬件复兴结束了。从 2003 年起，由于风冷芯片最大功耗和无法有效地开发更多指令级并行这两大孪生瓶颈，单处理器的性能提高速度下降到每年不足 22%（见图 4-1）。事实上，Intel 在 2004 年取消了自己的高性能单核处理器项目，转而和其他公司一起宣布：为了获得更高性能的处理器，应当提高一个芯片上集成的内核数目，而不是加快单核处理器的速度。这是一个标志着历史性转折的里程碑，处理器性能的提高从单纯依赖指令级并行(ILP)转向数据级并行(DLP)和线程级并行(TLP)。

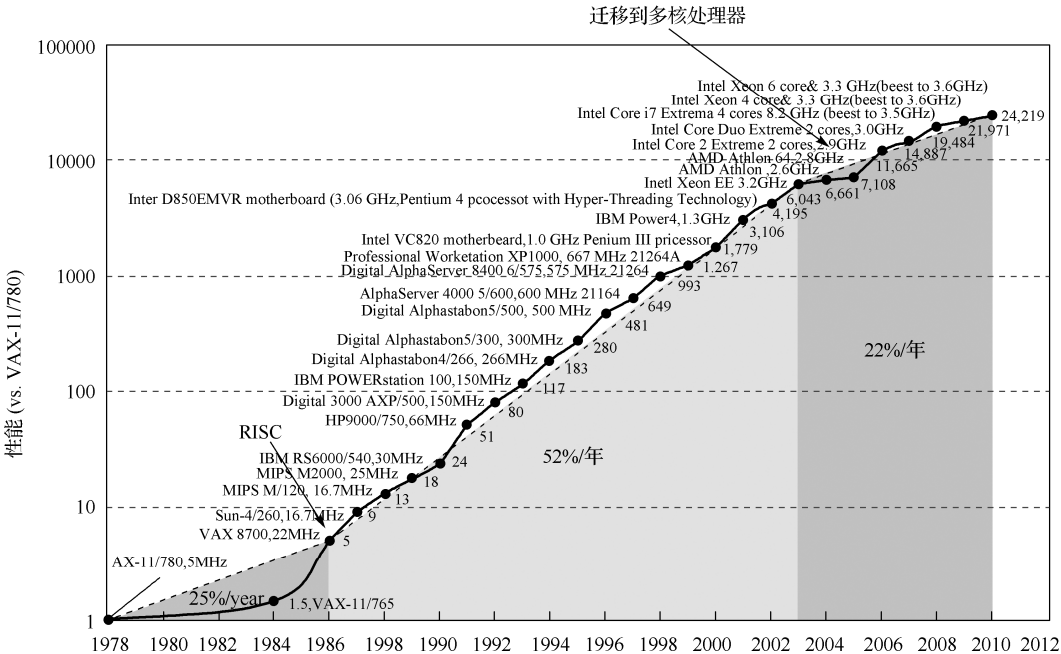


图 4-1 单核 CPU 性能提升自 2003 年起开始减缓

我们知道 CPU 的执行时间可以表示为：

CPU 执行时间 = 指令数 × 每条指令的执行周期（CPI）× 每周期时间

因此，为了优化 CPU 的计算性能（也就是减少 CPU 执行时间），可以主要从这三方面入手，如图 4-2 所示。

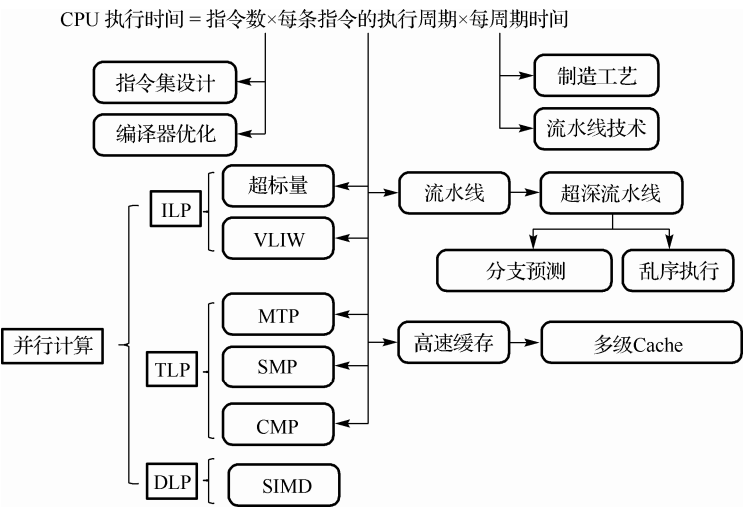


图 4-2 CPU 性能提升的主要技术脉络

首先是尽可能减少实现某特定功能的指令数，这又可以从两方面进行优化。第一是优化的指令集设计，让处理器的指令集尽可能适应所面向的应用，减少指令总数。比如，ARM 处理器的算术指令都支持第二操作数的移位操作；所有指令都支持条件执行；多 Load 多 Store 指令（LDM 和 STM 类指令）；等等。第二是通过优化编译器的设计，使得其生成的机器指令尽可能精简。

其次是尽可能降低 CPU 时钟的周期，也就是提高 CPU 的主频。这也可以主要通过两个手段获得。其一是不断降低 CPU 的制造尺寸，通过降低特征尺寸提高系统主频；其二是通过引入流水线技术，将一条指令的执行划分为可以独立进行的若干个阶段，也就是流水线的级数。理论上来说，划分的级数越多，每一级的工作就越简单，因此每一级的延时也就越小，由此系统的主频就可以越高。但是系统的主频提升不是没有止境的，这首先是因为工艺的提升是有物理限制的；其次，随着工艺的不断提升和主频的提高，由此而带来的问题也越来越多，比如由于工艺提升而造成的静态功耗问题和电压下降所造成的噪声容限问题，另外由于主频的提升对于系统全局同步时钟的设计也会带来极大的挑战；再次，流水线不可能总是处于理想的充满状态，流水线的级数越多，由于流水线被打断而造成的开销也就越大，为了解决超深流水线被打断的问题需要很多新的技术（下面将讨论这个问题），有文献指出超过 22 级以上的流水线已经没有性能收益。

提升 CPU 性能的第三条途径是降低每条指令的平均执行周期数（CPI，Cycle Per Instruction），或者提升每周期执行的平均指令数（Instruction Per Cycle，IPC）。正如前面所提到的，超深流水线被打断所产生的开销巨大，为了尽可能减少流水线被打断，分支预测（Branch Prediction）技术和乱序执行（OoO，Out of Order）技术被引入。另外，由于 CPU 速度的提升与外部存储器（主要是指动态存储器 DRAM）的差距越来越大，为了弥补这个速度差，高速缓存（Cache）已经成为现代高性能微处理器必不可少的部件，对于高主频处理器，仅仅通过一级缓存已经难以满足系统的需要，因此往往采用二级甚至三级缓存架构来逐级弥补处理器与外存的速度差。

除了上述两类技术外，提升每周期执行指令数的最有效方法莫过于并行化。根据并行化对象的不

同,我们可以将其分为指令级并行(ILP, Instruction Level Parallelizing)、任务级并行(TLP, Task Level Parallelizing)和数据级并行(DLP, Data Level Parallelizing)。如果说流水线技术是一种纵向的指令级并行技术,那么超标量技术和超长指令字(VLIW)技术则属于横向的指令级并行技术,后两者都是在一个时钟周期发射多条指令,由 CPU 内部并行的多条流水线同时执行多条指令,所不同的是超标量技术完全由硬件自动选择可以同时发射的指令,而 VLIW 技术则是在编译的过程中由编译器决定可以同时发射的指令,并将其合并为一条超长指令。任务级并行简单地说就是多核技术,通过操作系统的支持将任务划分为多个线程,并调度到多个同构或异构的处理器上并发运行。常见的任务级并行技术包括 SMP(对称多处理),对于将多个同构核集成在同一芯片的情况也被称为 CMP(Chip Multi Processor)。数据级并行最常见的技术就是 SIMD(Single Instruction Multiple Data)技术,该技术可以通过一条指令同时完成多个数据的相同运算。降低 CPI 所涉及的技术通常属于 CPU 的微架构设计,本节的后续部分将分别简单介绍这些常见的处理器技术。

4.1.2 复杂指令集(CISC)与精简指令集(RISC)

在微处理器的发展历史上,按微处理器的指令集特性可以将其分为两种主要架构:一种是 CISC(Complex Instruction Set Computer,复杂指令集计算机)架构,另一种是 RISC(Reduced Instruction Set Computer,精简指令集计算机)架构。Intel 体系架构的处理器从指令系统上看均为复杂指令集架构,ARM、MIPS 以及 PowerPC 均为精简指令集架构。复杂指令集的指令长度可变,指令的编码密度更高。精简指令集的指令长度是固定不变的,像是在 PowerPC 体系架构中,所有指令的长度均为 32 位。ARM 体系架构支持两种长度指令集,即 32 位的 ARM 指令集与 16 位的 Thumb 指令集(Thumb2 技术采用的是 16 位和 32 位混编指令集),但需要配置处理器,指定当前执行的指令集类型。相比较而言,16 位指令集可以有效提高代码密度,相同时间内,处理器载入更多指令以供执行。但是由于 16 位指令编码的空间有限,所以 16 位指令的功能可能较弱。

传统的 CISC 架构有其固有的缺点,即随着计算机技术的发展而不断引入新的复杂的指令,为支持这些新增的指令,计算机的体系结构会越来越复杂。然而,在 CISC 指令集的各种指令中,其使用频率却相差悬殊,大约有 20%的指令会被反复使用,占整个程序代码的 80%。而余下的 80%的指令却不经常使用,在程序设计中只占 20%,为了实现这些不常用的复杂指令,处理器硬件的复杂性以及由此消耗的能量却非常巨大,这不仅增加了处理器的硬件成本,同时由于这种硬件的复杂性使得处理器的主频也难以提高。显然,这种结构是不太合理的。

基于以上的不合理性,1979 年美国加州大学伯克利分校提出了 RISC 的概念。RISC 并非只是简单地减少指令,而是把着眼点放在了如何使计算机的结构更加简单合理地提高运算速度上。RISC 结构优先选取使用频率最高的指令,避免复杂指令;将指令长度固定,指令格式和寻址方式种类减少;以控制逻辑为主,不用或少用微码控制等措施来达到上述目的。由于硬件复杂性的降低,同时 RISC 处理器采用了流水线的微结构设计,使得同样制造工艺条件下 RISC 处理器的主频要更高。从理论上来说,这意味着 RISC 处理器在同等工艺条件下性能更高(当然为了达到这个目标,RISC 处理器的编译器需要做更多的优化工作以弥补指令集功能较弱的缺点)。

随着处理器技术的发展,CISC 与 RISC 架构的界限正在逐渐变得模糊,很多传统的 CISC 处理器,比如 Intel 的 X86 系列,为了提升主频,往往在 CPU 内部大量采用了 RISC 架构的技术。CISC 系列的 CPU 以 80x86 和 Motorola 68K 家族为代表,RISC 系列的 CPU 有很多家,比如 ARM、MIPS、PowerPC 等(国产 CPU 中的龙芯和众志也都采用 RISC 架构)。

归纳一下,RISC 架构和 CISC 架构微处理器比较,通常具有以下特征:

- ① 采用固定长度的指令格式,指令归整、简单,基本寻址方式有 2~3 种。

② 使用单周期指令，便于流水线操作执行。

③ 大量使用寄存器，数据处理指令只对寄存器进行操作，只有加载/存储指令可以访问存储器，以简化硬件的设计。

4.1.3 CPU 的流水线技术

CPU 的流水线技术是一种将每条指令分解为多步，并让各步操作重叠，从而实现几条指令并行处理的技术。程序中的指令仍是一条条地顺序执行，但可以预先取若干条指令，并在当前指令尚未执行完时，提前启动后续指令的前期操作步骤。这样显然可加速一段程序的运行过程。市场上推出的各种不同的 16/32 位微处理器基本上都采用了 CPU 的流水线技术。如 ARM 系列内核均使用了流水线技术，以六级流水线技术为例^①，CPU 的流水线技术的 6 步如下。

(1) 取指令。CPU 从高速缓存或内存中取一条指令。

(2) 指令译码。分析指令性质。

(3) 地址生成。很多指令要访问存储器中的操作数，操作数的地址也许在指令字中，也许要经过某些运算得到。

(4) 取操作数。当指令需要操作数时，就需要再访问寄存器堆，对操作数寻址并读出。

(5) 执行指令。由 ALU 执行指令规定的操作。

(6) 存储或“写回”结果。最后运算结果存放至某一内存单元或写回寄存器。

在理想情况下，每步需要一个时钟周期。当流水线完全装满时，每个时钟周期平均有一条指令从流水线上执行完毕，从平均的执行速度上看，相当于每个时钟周期执行一条指令。

流水线技术是通过增加计算机硬件来实现的。例如，要能预取指令，就需要增加取指令的硬件电路，并把取来的指令存放到指令队列缓存器中，使 CPU 能同时进行取指令和分析、执行指令的操作。因此，在 16/32 位微处理器中一般含有两个算术逻辑单元 ALU，一个主 ALU 用于执行指令，另一个 ALU 专用于地址生成，这样才可使地址计算与其他操作重叠进行。

在引入流水线时，应该确保流水线中的指令不会试图在相同时间使用同一个硬件资源，还需要确保不同流水级中的指令不会互相干扰，这都使得流水线的实现会相对复杂许多。关于这个问题，我们可以用 Cache 作为一个例子，设想在某个周期中，CPU 正在执行一条访存指令，同时流水线的取指逻辑还在取下一条指令。此时如果 CPU 采用指令数据统一的 Cache 就会造成冲突，而采用指令数据相分离的 Cache 结构（也就是所谓的哈佛结构）就可以解决这个问题，因为 CPU 可以同时访问数据 Cache 和指令 Cache。

流水化的主要效果是通过重叠指令的执行过程来改变它们的相对执行时间，提高 CPU 指令吞吐率，即单位时间内完成的指令数。但是有一些被称为冒险（Hazard）的情景，会阻止指令流中的下一条指令在其自己的指定时钟周期内执行。冒险会引起停顿，降低流水化所能获得的理想加速比，共有以下 3 类冒险。

(1) 结构冒险：当处理器以流水线方式工作时，指令的重叠执行需要实现功能单元的流水化和资源的多个复本，以允许在流水线中出现所有可能的指令组合。如果由于资源冲突而不能容许某些指令组合，就会导致结构冒险。例如，处理器可能仅有一个寄存器堆写端口，但在特定情况下，流水线可能希望在一个时钟周期内执行两个写操作，这就会发生结构冒险；我们在前面介绍的数据指令统一 Cache 的例子也能造成结构冒险。结构冒险在流水线中很常见，减少结构冒险引起的停顿能够显著提升 CPU 的性能。

① 实际上的流水线可能与这里的例子有很大不同。我们将在本章的 4.2.2 节详细介绍 ARM7TDMI 的三级流水线。

(2) 数据冒险：根据流水线中的指令重叠执行，指令之间存在先后顺序。如果一条指令取决于先前指令的结果，就可能导致数据冒险。根据指令中读、写访问的顺序，可以将数据冒险分为 3 类：RAW（写后读）、WAW（写后写）、WAR（读后写）。利用转发技术或者乱序执行技术可以将数据冒险引起的停顿减少。

(3) 控制冒险：分支指令及其他改变程序计数器的指令实现流水化时可能导致控制冒险。控制冒险会伤害流水线的性能，通常使用分支延迟（比如 MIPS 架构）及类似相对简单的机制来减少分支代价。当流水线越来越深、分支的潜在代价增加时，通常采用更高级的分支预测技术以及与之相关的推测执行技术（Speculative Excution）来减少分支代价。

流水线中的冒险会使流水线停顿，造成冒险的原因是因为指令之间存在一定的依赖关系。

要确定一个程序中是否存在多少并行以及如何开发并行，判断指令之间的相互依赖关系是至关重要的。如果两条指令相关（一条指令依赖于另一指令，因此“相关”的英文说法是 Dependency），它们就不是并行的，尽管它们可以部分重叠，但必须按顺序执行。共有 3 种不同类型的相关：数据相关（也称为真数据相关）、名称相关和控制相关。

(1) 数据相关。

如果以下任一条件成立，则说明指令 j 数据相关于指令 i：

- 指令 i 生成的结果可能会被指令 j 用到。
- 指令 j 数据相关于指令 k，指令 k 数据相关于指令 i。
- 数据相关也被称为真相关，因为后面的指令依赖于前面指令的执行结果。

(2) 名称相关。

当两条指令使用相同的寄存器或存储器位置（称为名称），但与该名称相关的指令之间并没有数据流动时，就会发生名称相关。在指令 i 和指令 j（按照程序顺序，指令 i 位于指令 j 之前）之间存在两种类型的名称相关。

- 当指令 j 对指令 i 读取的寄存器或者存储器位置执行写操作时就会在指令 i 和指令 j 之间发生反相关。简单地说就是所谓“读后写”（WAR），为了保证程序的正确执行，CPU 必须保证写操作在读操作之后完成（否则，前面读指令所得到结果就可能是后面指令写入的新值）。
- 当指令 i 和指令 j 对同一寄存器或存储器位置执行写操作时，发生输出相关。输出相关实际上是“写后写”（WAW），为了保证程序的正确，CPU 需要保证两条指令写入的顺序不能改变，否则目标地址中的内容可能是前一条指令写入的值。
- 名称相关在一个单发射顺序处理器上通常不会造成任何问题，但是对于一个多发射的超标量处理器，即使该处理器采用顺序执行，依然有可能造成冒险。这是因为两条名称相关的指令可能被发射到不同的功能单元同时执行，指令完成时间的不同可能造成后一条指令比前一条指令先完成。

(3) 控制相关。

控制相关决定了指令 i 相对于分支指令的顺序，使指令 i 按正确的程序顺序执行，而且只会在应当执行时执行。

指令 i 与指令 j 的相关性及其产生的冒险描述如表 4-1 所示。

表 4-1 指令的相关性及冒险

相关性	冒险
数据相关	数据冒险（RAW 冒险）
名称相关（反相关）	数据冒险（WAR 冒险）

续表

相关性	冒险
名称相关（输出相关）	数据冒险（WAW 冒险）
控制相关	控制冒险

如果两条指令是数据相关的，那么它们必须按顺序执行，不能同时执行或不能完全重叠执行。在现代处理器设计中，可以有两种不同的技术来减少数据冒险引起的停顿。第一是通过数据转发技术，将前一条指令在执行阶段产生的结果通过专门的通道提前转发给后面依赖这个结果的指令，该技术通常应用于单发射顺序处理器；第二是所谓的动态调度技术，也就是乱序执行技术，该技术允许就绪（不等待其他指令结果）的指令先发射执行，最大程度地使流水线充满。该技术通常应用于多发射（超标量）处理器。

由于没有在指令间传递值，名称相关并不是真正的相关，因此如果改变这些指令中使用的名称（寄存器号或存储器位置），使这些指令不再冲突，那么名称相关中涉及的指令就可以同时执行，或者重新排序。对于寄存器操作数，重命名操作更容易，称为寄存器重命名。寄存器重命名可以由编译器静态完成，也可以由硬件动态完成。通过寄存器重命名可以消除潜在的寄存器 WAR 冒险和 WAW 冒险。

冒险会使流水线停顿，导致处理器性能下降。一个流水化处理器执行每条指令占用的时钟周期数（CPI，Cycles per Instruction）等于基本 CPI 与因为各种停顿而耗费的全部周期之和：

流水线平均 CPI = 理想 CPI + 结构化停顿 + 数据冒险停顿 + 控制停顿

理想流水线 CPI 可以用来度量能够实现的最佳功能。通过缩短上述公式中右侧的各项，可以降低总流水线 CPI，也就是增加每个时钟周期完成的指令数（IPC，Instructions per Cycle）。根据上述公式，表 4-2 给出了流水线中的一些关键技术，并说明这项技术能够缩小总 CPI 的哪一部分，以此来刻画各种技术的特征。

表 4-2 流水线中研究的主要技术以及这些技术对 CPI 的影响

技术	降低 CPI 的哪一组成部分
转发和旁路	潜在的数据冒险停顿
延迟分支和简单分支调度	控制冒险停顿
基本编译器流水线调度	数据冒险停顿
基本动态调度（记分板）	由真相关引起的数据冒险停顿
循环展开	控制冒险停顿
分支预测	控制停顿
采用重命名的动态调度	由数据冒险、输出相关和反相关引起的停顿
硬件推测	数据冒险和控制冒险停顿
动态存储器消除二义	涉及存储器的数据冒险停顿
每个周期发出多条指令	理想 CPI
编译器相关性分析、软件流水线、踪迹调试	理想 CPI、数据冒险停顿
硬件支持编译器推测	理想 CPI、数据冒险停顿、分支冒险停顿

4.1.4* CPU 的分支预测技术

分支预测（Branch Prediction）是解决处理分支指令导致流水线失败的数据处理方法，由 CPU 来判断程序分支的进行方向，能够加快运算速度。当包含流水线技术的处理器处理分支指令时就会遇到一个问题，根据判定条件的真/假的不同，有可能会产生跳转，而这会打断流水线中指令的处理，因为处理器无法确定该指令的下一条指令，直到分支执行完毕。流水线越长，处理器等待的周期便越长，

因为它必须等待分支指令处理完毕,才能确定下一条进入流水线的指令,分支预测技术便是为解决这一问题而出现的。分支预测需要回答两个问题:第一是分支指令方向预测,就是预测分支指令是跳转(taken)还是不跳转(not taken);第二是分支指令跳转地址的预测,如果预测不跳转,处理器可以直接预取分支指令之后的指令,但是如果预测将发生跳转,处理器的预取逻辑必须知道跳转的目标地址才可以进行取指。但是通常情况下,目标地址只有在该分支指令进入执行阶段才能计算出来,为了避免预取硬件的等待,需要在该分支指令的译码阶段对目标地址进行预测。

我们可以将程序中的分支分为两种基本类型:第一种,也是最常见的,我们称为条件分支,我们用这类分支指令构建程序中的 if...else...语句和循环;第二种被称为无条件分支,这类分支指令总是发生跳转,其进一步又可以被细分为 3 类:① 立即分支,也就是常见的绝对跳转指令;② 间接分支指令,间接分支指令的目的地址是一个变量,该目的地址被存放在一个寄存器或者内存地址中,这类分支通常用于函数指针的调用,在采用面向对象的高级语言所生成的机器码中该类指令的比例明显增高;③ 函数返回指令,该指令在函数的结尾处被调用。对于无条件分支,因为它们总是跳转,因此对它们的方向预测比较简单,但是对于间接分支指令的目标地址预测则一直没有得到很好的解决。下面介绍的预测器技术重点将解决条件分支的方向预测和目标地址预测。

分支预测技术包含静态分支预测和硬件在执行时进行的动态分支预测。

最简单的静态分支预测方法就是任选一条分支或者固定选择一种情况,比如 Intel 的 80486 处理器就总是预测分支跳转,而 Sun 公司的 SuperSparc 则总是预测分支不跳转。另一种静态预测是假设向后跳转(PC 值减少)总是发生,而向前跳转(PC 值增加)则总是不发生,该算法也被称为 BTFN(Backward Taken Forward Not Taken),显然这种算法对于循环是有利的,HP7X00 采用该算法。更精确的办法是根据原先运行的结果进行统计从而尝试预测分支是否会跳转,这就是所谓的 Profile 方法。任何一种分支预测策略的效果都取决于该策略本身的精确度和条件分支的频率。然而,静态分支预测的正确率一般只有 40%~60%。

动态分支预测是近来的处理器已经广泛采用的技术。最简单的动态分支预测策略是分支预测缓冲区(Branch Prediction Buffer)或分支历史表(Branch History Table)。而对于目标地址的预测,一种常见的解决方法是采用分支目标缓冲区(BTB, Branch Targets Buffer),BTB 一般以 PC 的低位作为索引,表项中保存了对应 PC 地址处存放的分支指令上次 Taken 后的目标地址。这样,如果预测器判断一条分支指令将发生跳转,则通过 BTB 就可以得到其目标地址。下面将介绍几种常见的动态预测器。

1. 简单的 2 位预测器

最简单的动态分支预测机制是分支预测缓冲区或分支历史表。分支预测缓冲区是一个小型存储器,根据分支指令地址的低位部分进行索引。这个存储器中包含一位或数位,表明该分支最近是否曾发生过跳转(比如 1 表示跳转,0 表示不跳转)。采用这样一种缓冲区时,事实上并不知道预测是否正确,甚至它有可能是由另外一条具有相同低位地址的分支指令放入的(这时我们称这个现象为地址冲突)。这个预测只是一种提示,CPU 假定这个预测是正确的,并开始在预测方向上取指。如果这一预测是错误的,CPU 会将预测位进行修正后存回。

2 位预测器的每个缓存单元为两个比特位,用于预测分支是否被选中,该缓存单元(BPB)在 IF 流水线中使用指令地址进行访问。如果指令的译码结果为一条分支,并且该分支被预测为选中(Taken),则在 BTB 中得到目标地址之后立即从目标地址取指。否则,继续进行顺序提取和执行。如果预测错误,将有可能改变预测位。2 位预测器的有限状态机如图 4-3 所示。

许多分支被选中 and 不被选中的概率并非相等,而是严重偏向其中一种状态。在 2 位预测器中,使用 2 个位数对系统中的 4 种状态进行编码。当然我们也可以用更多的位数来描述分支的预测,比如采

用大于 2 的 n 比特。但是对 n 位预测器的研究已经证明，2 位预测器的效果几乎与 n 位预测器相同，所以大多数系统都采用 2 位分支预测器，而不是更具有一般性的 n 位预测器（多于 2 位的预测意味着更大的芯片面积和能耗）。由于我们尝试开发更多的 ILP，所以分支预测的准确度变得非常关键。而对于 SPEC 这样的基准测试程序，一个拥有 4000 项的 2 比特缓冲区，其性能大体和无限缓冲相当。图 4-4 所示是一个简单的 2 位预测器与目标地址缓冲区构成的取值逻辑。

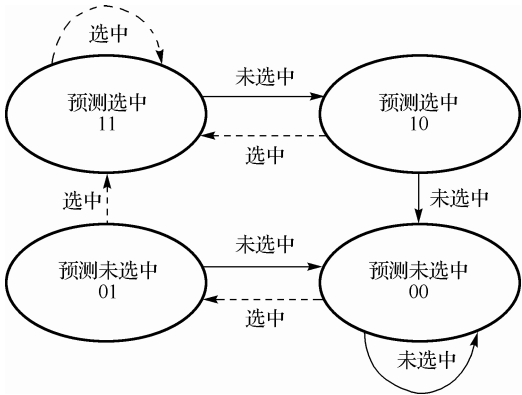


图 4-3 2 位预测机制中的状态

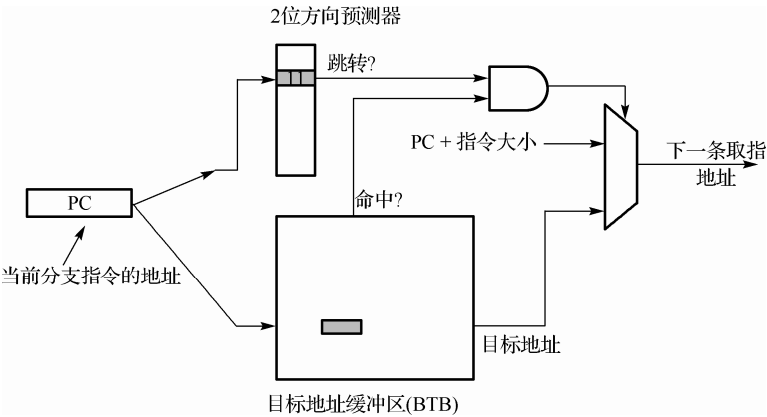


图 4-4 方向预测器与目标地址缓冲区构成的取值逻辑

2. 相关分支预测

2 位预测器仅适合用单个分支的最近行为来预测该分支的未来行为。如果同时还能查看其他分支的最近行为，而不是仅查看要预测的分支，那就有可能提高分支预测准确度。考虑如下一小段代码：

```
if (aa==2)                //branch1
    aa=0;
if (bb==2)                //branch2
    bb=0;
if (aa!=bb) {             //branch3
    :
}
```

将这三个分支分别标记为 branch1、branch2 和 branch3。由代码可以看出：分支 branch3 的行为与分支 branch1 和 branch2 的行为有关。显然，如果分支 branch1 和 branch2 都未执行转移（即其条件都

为真，且 aa 和 bb 均被赋值为 0)，aa 和 bb 明显相等，所以会进行 branch3 分支转移。如果预测器仅利用一个分支的行为来预测分支结果，那显然不会捕获到这一行为。

利用其他分支行为来进行预测的分支预测器称为相关预测或两级预测器。现有相关预测器增加了最近分支的行为信息，来决定如何预测一个给定分支。在一般情况下， (m,n) 预测器利用最近 m 个分支的行为从 2^m 个分支预测器中进行选择，其中每个预测器都是单个分支的 n 位预测器。这种相关预测器的吸引力在于它的预测率可以高于 2 位方案，而需要添加的硬件很少。最近 m 个分支的全局历史可以记录在 m 位的移位寄存器中，其中每一位记录是否执行了该分支指令，将分支指令的低地址与 m 位全局历史串联在一起，就可以对分支预测缓冲区进行寻址。相关预测器中的一个代表是所谓的 GShare 预测器，该预测器将一个全局的分支历史寄存器的值与 PC 值的低位进行异或后作为分支预测缓冲区的索引。其中，全局分支历史寄存器是一个固定长度（比如 m 位）的移位寄存器，其中记录了当前分支指令之前的 m 个分支的跳转情况。图 4-5 所示为 GShare 预测器的示意图。

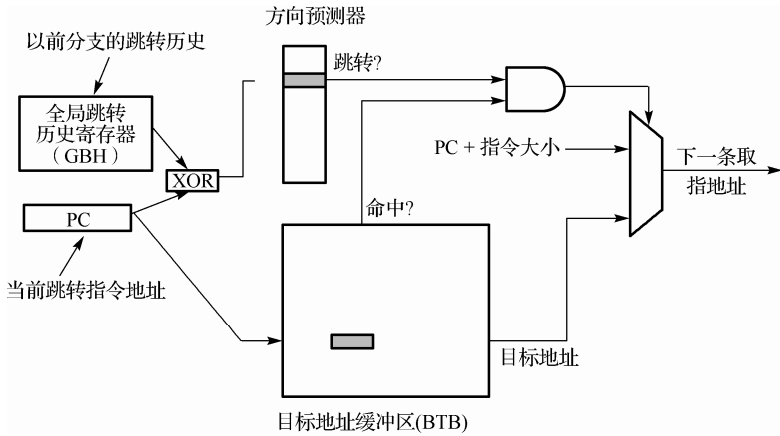


图 4-5 GShare 预测器

除了采用全局分支历史寄存器记录最近的分支跳转情况外，还存在一种通过记录特定分支指令的跳转历史来进行预测的分支预测器，我们称这种预测器为局部历史预测器，图 4-6 所示为局部历史预测器的示意图。

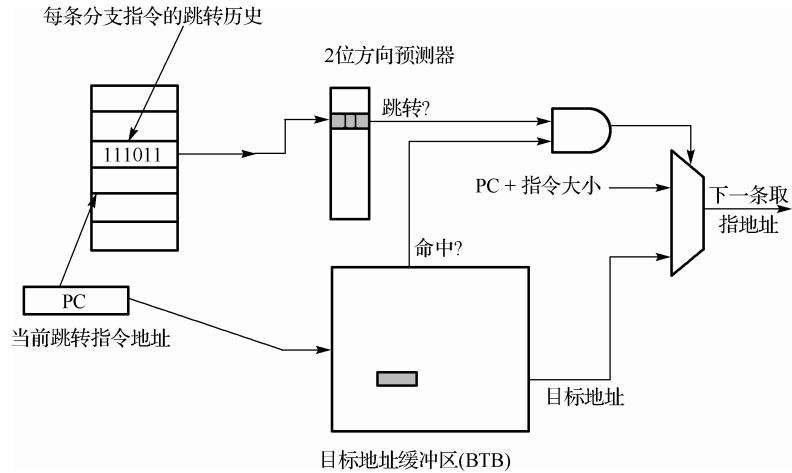


图 4-6 局部历史预测器

3. Tournament 预测器

采用相关分支预测器主要是因为观察到：仅适用局部信息的标准 2 位预测器无法预测某些重要分支，而通过增加全局信息就可能提升性能。Tournament 预测器更进一步，它采用了多个预测器，通常是一个基于全局信息的预测器和一个局部信息的预测器，并用选择器将它们结合起来，如图 4-7 所示，这也是为什么我们这类预测器为混合预测器的原因。Tournament 预测器既可以以中等规模的预测位（ $8\times2^{10}\sim32\times2^{10}$ 位）实现更好的预测准确度，还可以有效地利用超大量预测位。Tournament 预测器是一种整体局部自适应预测器。

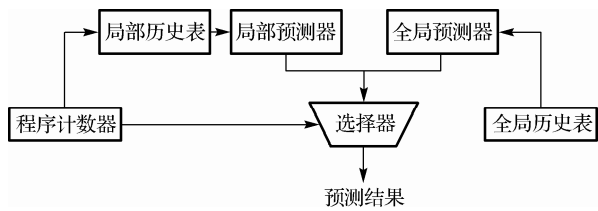


图 4-7 典型的 Tournament 预测器

现有 Tournament 预测器为每个分支使用一个 2 位饱和预测器，根据哪种预测器（局部、全局，或者两者的组合方式）在最近的预测中最为有效，从两个不同的预测器中进行选择。在简单的 2 位预测器中，饱和计数器要在两次预测错误之后才会改变优选预测器的身份。Tournament 预测器的优势在于它能够为特定分支选择正确的预测器，这一点是至关重要的。Tournament 预测器现已广泛应用在很多高性能处理器中。Alpha 21246 处理器中采用的就是这种混合预测器，该处理器将一个 1024 个局部跳转预测器与一个 4096 项全局预测器结合在一起，有兴趣的读者可以参阅扩展阅读[6]。

4.1.5* 乱序超标量处理器

乱序执行（out-of-order execution）是指 CPU 允许将多条指令不按程序规定的顺序分开发送给各相应电路单元处理的技术。这样将根据某个电路单元的状态和各指令能否提前执行的具体情况进行分析后，将能提前执行的指令立即发送给相应电路单元执行，在这期间不按规定顺序执行指令，然后由重新排列单元将各执行单元结果按指令顺序重新排列。采用乱序执行技术的目的是为了使 CPU 内部电路满负荷运转并相应提高了 CPU 运行程序的速度。一般乱序执行技术总是和超标量技术同时采用（事实上，我们没有办法在单发射的处理器上实现乱序执行）。

所谓超标量处理器是指在处理器内核中可以实现多条指令的并发处理，流水线技术使得多条指令可以串行地在处理器内部进行并发，而超标量技术则是多个指令在横向上同时执行，因此超标量技术也称为多发射。这种技术能够在相同的 CPU 主频下实现更高的 CPU 吞吐率（throughput）。超标量处理器可以顺序执行，也可以采用乱序执行技术。处理器的内核中一般有多个执行单元（或称功能单元），如算术逻辑单元、位移单元、浮点单元等。未实现超标量体系结构时，CPU 在每个时钟周期仅执行单条指令，因此仅有一个执行单元在工作，其他执行单元空闲。超标量处理器在一个时钟周期可以同时分派多条指令在不同的执行单元中执行，这就实现了指令级的并行。超标量体系结构可以视作 MIMD（多指令多数据）。以 ARM 公司设计的 Cortex A8 处理器为例，其包含有两个算术逻辑单元，处理单元的多实例化使得处理器可以在一个时钟周期内并发执行两条指令（至少在理论上如此）。超标量处理器的性能较传统的标量处理器有了很大提升，同时提高了功能单位的利用度，有效改进了能效比，当前处理器逐渐朝超标量处理器的趋势发展。需要注意的是，超标量处理器的执行速度通常多于每周期一条指令。但同时处理多条指令不见得就是超标量化，像流水线技术与多核技术，就是使用不同的方式

达到同时处理多条指令的目的。而超标量处理器的调度器从内存读取指令，并决定将哪些指令并行处理，将这些指令再调度到 CPU 内重复的功能单元。因而超标量处理器可以被想成多个平行的流水线，这些流水线可以平行处理一个执行序列中的指令。

图 4-8 是一个典型的乱序超标量处理器流水线设计。图中，F、D、R、D、I、C 分别代表 Fetch (取指)、Decode (译码)、Rename (重命名)、Dispatch (分发)、Issue (指令发射)、Commit (提交)。主要的微结构参数包括：层次化 Cache 结构相关参数、分支预测器相关参数、取指队列深度、发射缓冲深度、物理寄存器数目、ROB 容量、加载/存储队列深度、功能单元数目、MSHR (缓存缺失处理寄存器, Misses Status Handling Register) 容量等。

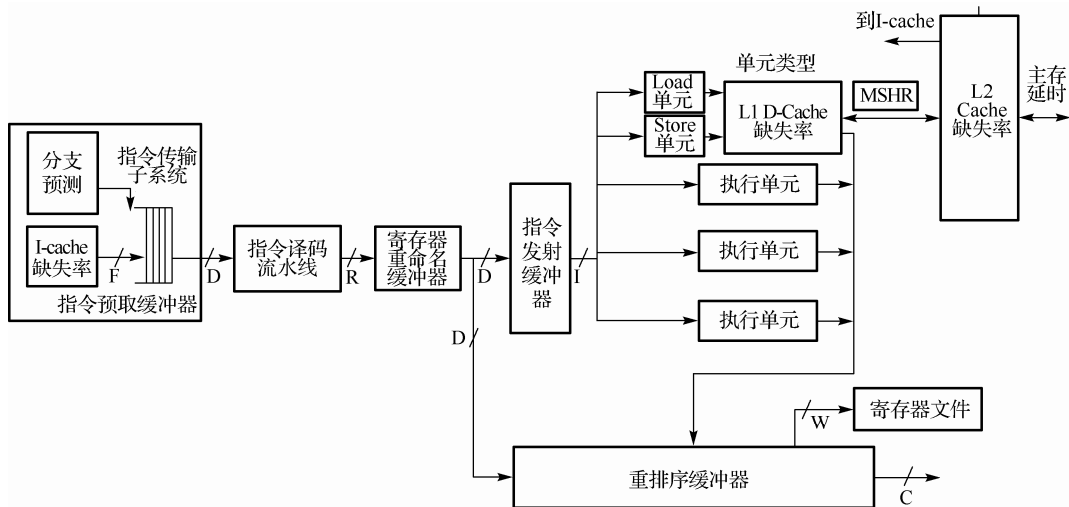


图 4-8 经典乱序超标量处理器结构框图

乱序执行微架构给现代处理器带来了性能上的显著提升。图 4-8 所示的乱序超标量处理器分为指令预取 (Fetch)、指令译码 (Decode)、寄存器重命名 (Rename)、指令发射 (Issue)、指令执行 (Execute)、写回 (Writeback)、指令提交 (Commit) 七级流水段。指令传输子系统包括指令预取、指令缓冲区和分支预测等单元，它为整个流水线提供持续的指令流。同时，该架构还用到了层次化缓存技术 (L1 D-Cache、L1 I-Cache、L2 Cache)，并且存在多个独立的功能单元以减少结构性冒险。

为实现乱序执行，在处理器的微结构设计中用到了很多关键技术。这些关键技术主要包括动态调度策略、硬件推测技术、寄存器重命名技术以及重排序缓冲区的使用。本节以图 4-8 所示的处理器体系结构为依托，对这些技术进行详细介绍，感兴趣的读者也可以参阅扩展阅读[1, 2, 4, 5]。

1. 动态调度的思想

在简单流水线中，如果流水线中的指令与要提取的指令之间存在数据相关，而且无法通过旁路或转发来隐藏这一数据相关，就会存在不可避免的冒险。冒险检测硬件会从使用该结果的指令开始，将流水线置于停顿状态。在清除这一相关之前，不会提取和发射新的指令。

简单流水线技术的一个主要限制是它们使用顺序指令发射与执行，如果一条指令停顿在流水线中，后续指令都不能继续进行。因此，如果流水线中两条相近的指令存在相关性，就会导致冒险和停顿。如果存在多个功能单元，这些单元也可能处于空闲状态。如果指令 j 依赖于长时间运行的指令 i (当前正在流水线中执行)，那么 j 之后的所有指令都必须停顿，直至 i 完成、 j 可以执行为止。例如，考虑如下代码：

```
mul  r1, r2, r4
add  r6, r1, r5
sub  r7, r3, r5
```

由于 ADD 对 MUL 的相关性（因为 ADD 指令的源操作数 r1 是 MUL 指令的目的操作数，这是一个典型的“写后读”相关）会导致流水线停顿，所以 SUB 指令不能执行，但是 SUB 与流水线中的任何指令都没有数据相关性。这一冒险会对性能造成限制，如果不需要程序顺序来执行，就可以消除这一限制。

在动态调度的方式中，硬件会重新安排指令的执行顺序以减少停顿，并同时保持数据流和异常处理的正确执行。当然，这是以硬件复杂度的显著提高为代价的。

在图 4-8 所示的处理器中，流水线允许同时执行多条指令，存在多个功能单元。指令经过 Rename 级之后将其按顺序发送（Dispatch）到发射缓冲区中。在发射（Issue）指令之前会检查所有的结构冒险，并等待数据冒险消失。如果发射缓冲区中存在一切就绪的指令（该指令的操作数都可用），并且相对的执行单元不存在结构冒险，那么这条指令立即发射出去，开始执行。这就是动态调度执行，有时我们也称其为乱序执行。动态调度的核心思想就是对于存放在发射缓冲区（有时也称为保留站，Reservation Station）中的指令采取“先就绪，先执行”的发射调度策略，而不是依据程序本身的指令顺序进行发射。

动态调度的处理器不能改变数据流，它只是在存在相关冒险时尽力避免停顿。

动态调度技术的实现需要有以下一些先决条件：第一，处理器必须支持多个功能单元，使得多条指令可以同时处于执行阶段。前面所举的例子中，如果乘法指令占用了一个 ALU，但是处理器没有其他的 ALU 的话，也没有办法将 SUB 指令提前执行，因为这时虽然规避了数据冒险，但是结构冒险又发生了。第二，在指令的执行流中存在长延时指令的执行，还是用上面的例子，如果乘法指令可以在一个周期内完成计算，并将结果提前传递给加法指令，则可以解决数据冒险的问题。但是如果乘法指令的执行时间比较长（通常对于浮点指令都需要较长时间，读操作 Cache 不命中也会造成长延时），通过数据转发已经无法避免流水线停顿。

2. 寄存器重命名

通过动态调度，乱序执行可以避免 RAW（写后读）冒险，但由于指令顺序的打乱，就有可能存在 WAR 和 WAW 冒险（读后写和写后写）。正如前面所介绍的，对于单发射顺序处理器而言，不会发生这两类冒险。但是对于超标量处理器，即使采用顺序执行机制也有可能产生 WAR 和 WAW 冒险，因为被分发到不同功能单元上的指令可能在不同的时间完成，从而无法保证程序原来的写顺序。WAR 和 WAW 冒险可以通过寄存器重命名来消除。对所有目标寄存器（包括较早指令正在读取或写入的寄存器）进行重命名，使乱序写入不会影响到任何依赖某一操作数较早值的指令，将 WAR 和 WAW 冒险降至最低。考虑如下可能会出现 WAR 和 WAW 冒险的代码序列：

```
sub  r1, r2, r5
add  r6, r1, r4
mul  r5, r3, r2
mov  r6, r4
```

以上代码中，SUB 与 MUL 之间存在一处反相关（WAR 冒险），ADD 与 MOV 之间存在一处输出相关（WAW 冒险）。这两处名称相关都可以通过寄存器重命名来消除。为简单起见，假定存在两个临时寄存器 T 和 S。利用 T 和 S，可以对这一序列进行改写，消除名称相关，如下所示：


```
sub  r1, r2, r5
add  r6, r1, r4
mul  T, r3, r2
mov  S, r4
```

此外，对于 r5 和 r6 的后续使用都必须使用 T 和 S 来代替。这就是寄存器重命名的基本原理。

在图 4-8 所示的处理器中，寄存器重命名工作由 Rename 流水级完成，而附加寄存器由处理器内部的物理寄存器来提供，以消除 WAR 和 WAW 冒险。重命名过的指令将被 Dispatch 到发射缓冲区中。

在我们理解所谓乱序执行机制时，必须明白乱序执行并不是目的，乱序执行其实是动态调度的结果。之所以要进行动态调度，是因为指令之间存在相关性（这里所说的相关性其实是真数据相关，也就是存在 RAW 冒险。而不是反相关和输出相关，本质上后两者如果是顺序执行，没有任何问题。但是在乱序后可能引起 WAR 和 WAW 冒险）。需要动态调度的另外一个原因是引起数据相关的前一条指令可能需要比较长的执行时间（比如浮点运算指令，以及访存指令），这种长延迟没有办法通过 Forwarding 和 Bypassing 机制进行掩盖，传统的顺序流水线没有办法解决这个问题，只能暂停流水线，停止发射所有后续指令，直到这条长延时指令完成执行并回写目的寄存器后，才能重新开始流水线的运行。乱序处理器在发现存在数据相关的指令序列后，将继续发射后续的指令。本质上来说，乱序处理器并不是发现了数据相关，而是在每个不同功能单元的保留站中分别察看是否有指令所需的所有操作数已经就绪，如果就绪则执行该指令，否则，该指令将停留在保留站中直到所需的操作数就绪。采用指令保留站的另外一个好处是自然地解决了由于潜在的乱序执行而造成的 WAR 和 WAW 冒险。因为在保留站中的指令将拥有自己独立的源操作数和目的操作数，所有针对原指令的寄存器的读写现在都变成了对保留站中临时寄存器的读写，只有在指令最终提交时才回写到寄存器堆。这种采用保留站暂存待发射指令，并利用其实现数据跟踪的动态调度算法也称为 Tomasulo 算法。Tomasulo 算法会跟踪指令的操作数何时可用，将 RAW 冒险降至最低，并通过对寄存器进行有效的动态重命名来处理反相关和输出相关。

通过以上分析，我们还可以得出这样一个结论：多个功能单元似乎是动态调度的前提，如果处理器只有一个功能单元（比如一个整数 ALU）是没有办法进行动态调度的，因为存在结构冒险。而当我们拥有多个并行的功能单元时，比如两个整数 ALU 和两个浮点 ALU 再加一个 Load-Store 队列，理论上可以在横向上（而不是流水线的纵向）同时执行 5 条不同的指令，也就是可以有 5 条指令同时处于执行阶段。

3. 推测执行技术以及重排序缓冲区

当尝试开发更多指令级并行时，控制相关的维护就会成为一项不断加重的负担。硬件推测执行（Speculative Execution）机制对采用动态调度的分支预测进行了一种虽细微但很重要的扩展。通过推测技术，取指、发射和执行指令，并且假设分支预测总是正确的，然后通过一些机制来处理推测错误的情景。

基于硬件的推测结合了 3 种关键思想：①用动态分支预测选择要执行哪些指令；②利用推测，可以在解决控制相关问题之前执行指令（能够撤销错误推测序列的影响）；③进行动态调度，以应对基本模块不同组合方式的调度。与之相对，没有推测的动态调度需要先解析分支才能执行后期基本模块的操作。

为了使处理器支持推测技术，在流水线中新增加了一个附加步骤，称为指令提交（Commit）。当一条指令不再具有不确定性时，才允许它更新寄存器堆或存储器。实现推测的关键思想在于允许指令乱序执行，但强制它们顺序提交，以防止在指令提交之前采取任何不可挽回的动作。因此，当添加推

测时，需要将指令执行的过程与指令提交区分开来，这是因为指令执行完毕的时间可能远远早于它们做好提交准备的时间。在指令执行序列中添加这一提交阶段需要增加一组硬件缓冲区，用来保存还没有提交的指令结果。这一硬件缓冲区称为重排序缓冲区（ROB，Reorder Buffer），也可用于在可被推测的指令之间传送结果。

重排序缓冲区提供了附加寄存器，ROB 会在一定时间内保存指令的结果，并为其他指令提供操作数，这段时间从完成该指令的相关运算起，到该指令提交完毕为止。如图 4-9 所示，重排序机制类似于一种旁路机制，这样可将在重排序缓冲区中保存的某个指令的结果送到等待它的功能单元中，不用经过数据寄存器。

相比提供操作数，重排序缓冲区另一个最大的作用是保证指令有序提交和异常行为恢复执行。图 4-8 所示的处理器中，寄存器重命名之后，将其按顺序分别分配到发射缓冲区和重排序缓冲区中。当发射缓冲区中指令的操作数可用，并且不存在结构冒险时，立即发射到功能单元进行指令运算，并将结果写到 ROB 中。当一个指令到达 ROB 的头部并且其结果出现在 ROB 中时，则进行正常提交。此时，处理器用结果更新寄存器和存储器，并从 ROB 中清除该指令。当发生异常或预测错误的分支到达 ROB 头部时，它指出推测是错误的。ROB 被刷新，执行过程进入异常行为处理或者从该分支指令的后续正常指令重新开始。由于在提交该指令之前寄存器值和存储器值都没有实际写入，所以在发生异常或分支预测错误时，处理器可以很轻松地撤销其推测操作。如果对该分支的预测正确，则该分支完成提交。而指令一旦提交完毕，它在 ROB 的相应项将被收回，寄存器或存储器被更新，不再需要 ROB 项。如果 ROB 填满，那么只需要停止将指令分配到 ROB 和发射缓冲区中，直到有空闲项目为止。ROB 窗口的存在保证了指令在乱序执行过程中被顺序提交。

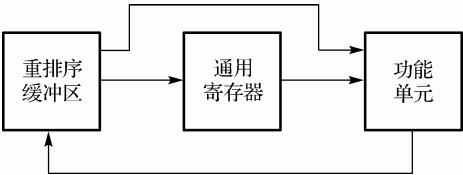


图 4-9 重排序缓冲区提供操作数

4.1.6* SIMD 和向量处理器

20 世纪 60 年代提出了并行硬件的另一种分类方法，并且一直沿用至今。该分类基于指令流的数量和数据流的数量。常规的单处理器具有单一的指令流和单一的数据流，而常规的多处理器具有多个指令流和多个数据流。这两种类别分别称为 SISD 和 MIMD。

在 MIMD 计算机上可以编写独立的程序并运行在不同的处理器上，而且这些程序可以协同完成一个共同的大型目标。但是编程人员通常仅编写单一程序，将其运行在 MIMD 计算机的所有处理器上，并使用条件语句控制不同的处理器执行不同的代码段。这种风格被称作单程序多数据（SPMD），它是 MIMD 计算机编程的正常方式。

SIMD 计算机对向量数据进行操作。例如，一个单一的 SIMD 指令可以把 64 对操作数相加，只需要把 64 对源数据发送到 64 个 ALU，就可以在一个时钟周期内得到 64 个和。

SIMD 的优点是所有并行执行单元都是同步的，它们都对源自同一程序计数器（PC）的同一指令做出响应。从程序员的角度来看，非常接近熟悉的 SISD 编程模型。尽管每个单元都执行相同的指令，但是每个执行单元都有自己的地址寄存器，这样每个单元都有不同的数据地址。因此，一个顺序应用程序编译后可能运行于串行硬件上并按 SISD 组织，也可能运行于并行硬件上按 SIMD 组织。

SIMD 的初衷是在几十个执行单元之间均摊控制单元成本（因为只需要一套指令解码和控制单元）。另外一个优点是降低程序存储器的大小——SIMD 只需要同时执行代码的一个副本，而消息传递的 MIMD 可能需要在每个处理器都有一份副本，共享存储器 MIMD 需要多个指令缓存。

SIMD 在使用 for 循环语句处理数组时最为有效。可以用一条指令同时（并发）实现对大量相

同结构的数据进行操作，因此，SIMD 是一种数据级并行（DLP，Data Level Parallelism）技术。SIMD 在使用 case 或 switch 语句时效率最低，因为此时每个执行单元必须根据不同的数据执行不同的操作。

SIMD 的一个更加古老和优雅的称呼是向量体系结构，它几乎等同于 Cray 公司制造的计算机。向量机结构与具有大量数据并行的问题非常匹配。除了具有 64 个 ALU 可以同时计算 64 次加法以外，与早期的阵列处理器类似，向量体系结构将 ALU 流水化，从而在低成本的情况下获得高性能。向量体系结构的基本理念是从存储器中收集数据元，并将它们按顺序放在一组寄存器中，然后在寄存器中对它们依次操作，最后将结果写回存储器。向量体系结构的关键特征是拥有一组向量寄存器。这样，向量体系结构可能拥有 32 个向量寄存器，每个寄存器包含 64 个 64 位宽的数据元。

与常规的标量体系结构指令集相比，向量指令具有几个重要的属性：

- 一条向量指令就指定了大量需要完成的工作——它等价于执行一个循环。因而对取指和译码带宽的需求显著降低了。
- 通过使用向量指令，编译器或程序员隐含指明向量中每个结果的计算与同一向量中其他结果的计算是不相关的，因而硬件无需检查一条向量指令内的数据相关。
- 相对于 MIMD 多处理器，包含数据级并行的应用程序采用向量体系结构和编译器能够更加容易地编写高效代码。
- 硬件只需在两条向量指令之间对每个向量操作数检查一次数据相关，而不是对向量内每个数据元检查一次。相关检查次数的降低也会使得功耗降低。
- 访问存储器的向量指令具有确定的存取模式。如果向量的每个元素都是地址连续的，那么从一组交叉存储器组中取回一个向量将会很快。因此，对整个向量而言，主存延迟的开销看上去只有一次，而不是对向量中每个字都有一次。
- 因为整个循环用具有预定义行为的向量指令所替换，循环转移所引起的控制相关就不存在了。
- 节省的指令带宽和相关检查以及存储器带宽的有效使用，使得向量体系结构在功耗方面优于标量体系结构。

由于这些原因，在同样的数据量前提下，向量操作比一组标量操作序列更快，并且如果应用程序可以频繁使用这些向量操作，就会促使设计者加入向量单元。

最早的 SIMD 或者向量机技术是应用于面向数学计算的巨型机或大型机系统的，正如前面提到的 Cray。后来，大多数专门面向数字信号处理的 DSP 处理器广泛采用了该技术，可以想象 SIMD 技术被应用于大量矩阵处理时的高效。桌面系统中首次出现该技术可能可以追溯到 Intel 公司的 MMX 技术，该技术为增强桌面处理器处理多媒体数据时的效率而引入了 SIMD 扩展指令集。随着媒体处理（包含 UI 渲染时的图像处理）在移动智能终端上的广泛应用，ARM 公司也在其系列处理器中引入了 NEON 技术，NEON 处理器其实就是一个面向媒体运算优化的 SIMD 计算引擎。在 Android 操作系统中的 Skia 图形库中，就存在大量的采用 NEON 指令进行优化的代码。

4.1.7* VLIW 处理器

超长指令字（VLIW，Very Long Instruction Word）指的是一种被设计为可以利用指令级并行（ILP）优势的 CPU 体系结构。一个按照顺序执行指令的非超标量处理器不能充分地利用处理器的资源，有可能导致低性能。性能可以通过同时执行一系列指令中的不同子步骤来提高（这就是流水线技术），或者像超标量架构一样，甚至完全并行执行多条处理器指令。进一步的提高可以通过指令的执行顺序与在程序代码中出现的顺序不同来实现，这就是乱序执行。这三种技术都要付出代价——增加了硬件的复杂性。在并行执行任何操作之前，处理器必须确认这些指令间没有相互依赖。例如，第一条指令的结

果作为第二条指令的输入，显然这样的两条指令无法同时执行，并且第二条指令不能先于第一条指令执行。乱序执行处理器增加了硬件资源用于调度指令和确认指令间的相互依赖。

VLIW 技术通过另一种方法来实现并行。VLIW 的并行指令执行是基于一个确定的调度。这个调度是程序在编译时就已经确定好的。由于决定乱序执行的工作是由编译器来完成的，因此处理器不再需要上面 3 种技术所需的调度硬件。结果 VLIW 处理器相比其他多数的超标量处理器提供了更加强大的处理能力但是更少的硬件复杂性（编译器的复杂性提高了）。正如一些其他比较新颖的架构，这种并行执行的概念只有当编译器能生成有效的代码时才变得有用。如果编译器无法准确地找出相关的代码并且生成可以利用 CPU 能力的目标代码，这些特殊目的的指令将变得毫无用处。

有两个原因使得 VLIW 技术一直没有能够成为主流：①VLIW 的指令集很难与以前的处理器兼容，由于在 VLIW 处理器中往往需要设计足够宽的功能单元，这使得它很难与前面开发的处理器相兼容；②由于 Cache 以及 DDR 存储器的存在，使得访存操作的延时难以预估，这使得基于编译器的静态调度非常困难。

4.1.8* EPIC 处理器

显示并行指令计算（EPIC, Explicitly Parallel Instruction Computing）是一种指令集架构，由 HP 和 Intel 联合开发。EPIC 允许处理器根据编译器的调度并行执行指令而不增加硬件复杂性，该架构由超长指令字架构发展而来，并做了大量改进。其指令中有 3 位是用来指示上一条运算指令是不是与下一条指令有相关性，是不是要等上一条指令运行完毕后才能运行下一条。如果没有相关性，则两条指令可同时由不同的 CPU 节点来处理，这种方式大大提高了 CPU 并行运算的效率。EPIC 成为 IA-64 架构的基础（IA 代表 Intel Architecture，即英特尔架构，与 IA-32 对应），这是英特尔与惠普共同开发的纯 64 位微处理器。英特尔的安腾（Itanium）系列处理器采用了这种架构。

4.2 ARM 内核

4.2.1 ARM 介绍

1. ARM 发展历史

1987 年一家叫 Acron Computers 的英国公司生产了一款类似于 Apple II 的个人计算机，称为 BBC Micro，当时在英国大为畅销。BBC Micro 与 Apple II 一样采用 6502 位 CPU，但紧接着 Acron Computers 着手研发其专用 32 位微处理器以应用于下一代的个人计算机当中，而这款专用的 32 位微处理器便被取名为 Acorn RISC Machine，简称 ARM，这款微处理器便是 ARM 处理器内核的前身。

1990 年 11 月由 Acorn、VLSI Technology、Apple 等厂商与创业者 Nippon Investment & Finance 合资成立了一家叫作 Advanced RISC Machine 的公司（简称 ARM 公司），专门从事微处理器研发与市场规划，但不涉足微处理器制造，其公司定位与 MIPS Technology 相当类似。之所以需要单独成立一家公司，是因为 Apple 希望该公司为其正在研发的全球首款 PDA-Newton 设计全新的处理器。早期 Apple 投入 300 万美金拥有了 ARM 公司 43% 的股份，1998 年，ARM 公司在英国和美国同时上市后，Apple 逐渐卖出了这些股份。2010 年 6 月中，苹果公司表示有意以 80 亿美元的价格收购 ARM 公司，但遭到拒绝。

Apple Computer 选择 ARM 架构来设计其个人数字助理（PDA, Personal Digital Assistant）产品，继 Newton Message Pad 100 型与 110 型之后，ARM 微处理器便声誉鹊起，在市场上占有了一席之地。

在 Newton Message Pad100 与 110 这两款应用中，Apple Computer 使用 ARM610 与 Apple 自行研发的 ASIC 共同担当核心处理器，虽然 Newton 系列在市场上并未大放异彩，仅销售 30 万台左右，但已对 PDA 市场做出了很好的示范，并且为 ARM 微处理器的应用提供了极佳的信誉。

另一个应用 ARM 的例子是美国的游戏制造公司 3DO，其在 ARM 微处理器的应用范例中也是十分突出的。这家公司的高性能游戏平台（称为 Panasonic REAL Multiplayer）已被授权给 Sanyo、Goldstar 与 Matsushita，且至少销售 50 万台以上。虽然 3DO 在其新一代的游戏机中改用 Power PC 当作 CPU，但仍然采用 ARM 微处理器来控制其系统的相容性。

在 1995 年 2 月，ARM 公司宣布与 Digital Semiconductor 公司合作发展新一代高性能微处理器。称为 StrongARM。最初 StrongARM 的生产是在 Hudson 的 Fab 厂，采用 0.35μm，三层金属制作。同年 ARM 亦宣称将发展一种具有创新性的微处理器架构，称为 Thumb。这种技术可降低程序代码的大小。因为在嵌入式核心的应用中，程序长短通常意味着系统所需存储与执行程序的存储体大小。因此降低程序代码长度便意味着降低嵌入式内核应用的成本，Thumb 便是应这种需求而发展起来的技术。更明确地说，Thumb 是 ARM 指令压缩形式的子集，可被包含在用户指定的 ASIC 或视为标准 ARM 微处理器的一部分。必须注意的是 Thumb 只相容于 ARM7 与 ARM7 之后的 ARM 微处理器架构。

在 Thumb 技术推出 18 个月之后，ARM 于 1996 年年底宣称 Piccolo 技术已发展成功。Piccolo 的出现大大加强了 ARM 架构下的数字信号处理能力，增加了许多 DSP 特性，有助于 ARM 系列在无线可携式产品（如移动电话与高档传呼机等）的应用广度。

ARM 公司作为知识产权供应商，本身不直接从事芯片生产，靠转让设计许可由合作公司生产各具特色的芯片，世界各大半导体生产商从 ARM 公司购买其 ARM 微处理器核授权，根据各自不同的应用领域，加入适当的外围电路，从而形成自己的 ARM 微处理器芯片进入市场。目前，全球 150 多家半导体公司都使用 ARM 公司的授权，这使得 ARM 技术获得更多的第三方工具、制造、软件的支持，又使整个系统成本降低，使产品更容易进入市场被消费者所接受，更具有竞争力。至 2008 年年底，基于 ARM 处理器的设备累计总出货量突破 100 亿台；2013 年年底，累计超过 520 亿颗基于 ARM CPU 的芯片被交付；2014 年全球手机数首次超过全球人口，而这些手机中的 90%采用基于 ARM CPU 的芯片。

到目前为止，ARM 已授权许多公司使用其 ARM 内核应用在不同领域，其中比较有名的合作伙伴如表 4-3 所示。

以 AKM 而言，它把 ARM 内核应用在其电信产品中，而 Cirrus Logic 则利用其在芯片组的经验来生产内含 ARM 微处理器的逻辑元件与周边装置。GEC Plessey 则把 ARM 微处理器的芯片应用于电信与无线数据市场。Sharp 类似于 AKM，主要是把 ARM 用在自家的产品中（Sharp 是 ARM 最早的客户之一，遗憾的是现在 Sharp 的半导体部门已经卖给 NXP 了），而 VLSI Technology 和 TI 则双双扮演 ASIC 设计服务的角色，当部分厂商需要使用 ARM 时，便可向其提出服务需求，由 VLSI Technology 或 TI 来协助产品设计。

表 4-3 ARM 主要合作伙伴以及主要商业模式

IP 供应商	硅芯片被授权方	原设备制造商	产品
ARM	英特尔 Strong ARM XScale	华为 惠普 联想	网络产品 IPAQ XP100
ARM	德州仪器 Baseband OMAP	诺基亚 联想	全球 80%的 GSM/3G 手机

续表

IP 供应商	硅芯片被授权方	原设备制造商	产品
ARM	高通 CDMA	中兴 海尔 大唐	全球 99% 的 CDMA 手机
ARM	摩托罗拉 Dragonball MX1, MX21	Palm	Palm 及其他 PDA
ARM	Atmel AT91 微控制器	众多 OEM	嵌入式产品
ARM	其他的合作伙伴主要有 AKM、Cirrus Logic、GEC Plessey、Sharp 等		

代表厂商	苹果	三星	MTK	高通	我国企业
授权方式	• 架构授权	• 软核授权	• 软核授权	• 高端架构+低端软核	• 软核/硬核
定制性	• 扩展指令 • 设计处理器微结构 • 后端设计	• 配置微结构参数 • 后端设计	• 配置微结构参数 • 后端设计	• 设计处理器微结构 • 后端设计	• 使用参考设计 • 后端设计 • 或嵌入硬核
市场	• 高端手机和平板	• 高端手机和平板	• 中低端手机	• 高中低全面覆盖	• 中低端手机和平板
生态	• 设计操作系统 • 定制 CPU 适配层 • 软硬适配	• 参与 Android 操作系统设计 • SoC 驱动开发	• 使用官方 Android 操作系统 • SoC 驱动开发	• 参与 Android 操作系统设计 • 定制 CPU 适配层 • SoC 驱动开发	• 使用官方 Android 操作系统 • SoC 驱动开发
特点	• 软硬协同 • 产业链整合 • 自建生态	• 产业链整合	• SoC 芯片整合	• 基带优势带动 • 组建基于高通生态环境	• 低价方案 • 依赖官方系统
紧跟 ARM 发布	• 独立演进微结构 • 按需扩展指令	• 快速根据 ARM 最新 CPU，形成标志 SoC	• 等待 CPU 完成市场检验，后发致人，多款覆盖全系列	• 高端独立演进 • 低端采用软核硬化 • 部分软硬适配	• 等待一线厂商推出最新标志性 SoC，再同质化跟进

ARM 体系结构目前被公认为是业界领先的 32 位嵌入式 RISC 微处理器结构，所有 ARM 处理器共享这一体系结构，因而确保了开发者转向更高性能的 ARM 处理器时在软件开发上可以得到最大的回报。伴随着 ARM 系列微处理器的不断发展，ARM 体系结构也由过去的 v4T 发展到了 v5TE、v5TEJ、v6、v7，v8。表 4-4 列出了 ARM 体系结构版本的特点及对应的 ARM 内核。

表 4-4 ARM 体系结构版本的特点及对应的 ARM 内核

ARM 体系结构版本	特点	对应的 ARM 内核
v4		SA-110、SA-1110（StrongARM）
v4T	支持 Thumb 指令集	ARM7TDMI、ARM720T、ARM9TDMI、ARM920T
v5TE	支持 ARM/Thumb 的交互工作、饱和路径 DSP 乘加指令	ARM1020E、Xscale、ARM9E-S、ARM966E-S
v5TEJ	支持 Java 加速器 Jazelle	ARM9EJ-S、ARM926EJ-S、ARM7EJ-S、ARM1026EJ-S
v6	SIMD 指令 TurstZone 技术	ARM1136EJ-S
v7	支持 Thumb-2 技术 NEON 协处理器	ARM Cortex 系列

ARM 第一款最成功的处理器内核 ARM7TDMI 基于 ARMv4T 架构的(T 表示支持 Thumb 指令)；ARMv5TE 架构添加了“服务于多媒体应用增强的 DSP 指令”，其代表处理器是 ARM926EJ。ARMv6 架构的代表是 ARM11 系列 CPU，主要包括 ARM1136J（F）-S 以及 ARM1176JZ（F）-S。ARM v6 是

ARM 进化史上的一个重要里程碑,从那时起,许多突破性的新技术被引进,存储器系统加入了很多崭新的特性,单指令流多数据流(SIMD)指令也是从 v6 开始引入的。v7 架构最重要的突破是引入了专门的 SIMD 引擎——Neon。与 v6 架构的 SIMD 指令依然采用主 CPU 的 ALU 进行计算不同,Neon 是独立于主 CPU 的协处理器,具有独立的流水线、运算单元和寄存器堆。另外,TrustZone 技术的引入也为更高安全需求的应用提供了保障。v7 架构一改以前处理器以数字作为命名的方式,而统一称为 Cortex 系列,该系列处理器又分为 3 个不同的应用领域,分别是:

- A 系列(Application):设计用于高性能的“开放应用平台”,越来越接近计算机。
- R 系列(Real Time):用于高端的嵌入式系统,尤其是那些带有实时要求的,既要快又要实时。
- M 系列(Microcontroller):用于深度嵌入式的、单片机风格的系统中。

2. ARM 架构的特点

在最初接触 ARM 时往往会觉得 ARM 非常“怪异”,有很多东西和我们以前学的 X86 体系不太一样,产生这样印象的主要原因是因为 ARM 属于比较纯粹的 RISC 架构,相比 CISC 架构的代表 x86 系列而言,ARM 在继承发扬 RISC 核心设计哲学的基础上又做了进一步的创新与扬弃。因此,即使在 RISC 阵营中,ARM 架构依然算得上是比较有特色的架构,本小节将试图给读者一个总的概述。首先介绍 ARM 作为经典 RISC 架构的主要特点。

1) 非常多的通用寄存器

ARM7TDMI 内核中一共包括 37 个通用寄存器,这与传统的 CISC 机器有非常大的区别,当然这也是 ARM 作为 RISC 处理器的一个非常重要的特征,寄存器多了就可以尽量减少对存储器的访问(比如传递参数、临时变量等)。虽然在一个 CPU 状态下,我们能够看到的寄存器最多只有 18 个(r0~r15、CPSR、SPSR),但是 ARM 的寄存器堆确实较 X86 或者 68K 要多了不少。

2) Load/Store 体系

与大多数的 RISC 架构类似,ARM 是典型的 Load Store 体系架构,也就是说指令集中除了 Load 和 Store 两类指令是可以访问存储器的,其他所有指令的源操作数和目的操作数都是存放在寄存器中的。CPU 硬件除了取指和执行这两类指令外不会访问存储器,甚至发生函数调用和中断响应时的返回地址都是由硬件自动保存到 r14 寄存器(中断响应还需要硬件自动将 CPSR 保存到相应的 SPSR 中),用户必须通过自己的软件中的 Store 指令才能将 r14 和 SPSR 中的值保存到堆栈中。

3) 没有硬件堆栈

因为是比较纯粹的 Load Store 架构,与我们以前学习的 CISC 机器不同,ARM 的处理器本身在硬件上没有堆栈的概念。这么说的意思是 ARM 本身在硬件上不会自己维护堆栈。比如,在发生函数调用的时候,返回地址是由硬件自动保存在 R14 寄存器中的,硬件不会将返回地址自动压栈。在中断或异常的时候,硬件也只是将返回地址保存到 R14, CPSR 保存到相应状态的 SPSR 中。如果用户需要函数嵌套调用或是中断嵌套的话,必须自己通过软件管理自己的堆栈。因此,理论上,在 ARM 架构下,程序员可以采用任何形式的堆栈组织方式(比如满递减栈、满递增栈、空递减栈和空递增栈),虽然绝大多数情况下,大家都按惯例采用满递减栈的方式。

4) 多种处理器工作模式及影子寄存器

ARM 处理器有多达 7 个的处理器工作模式^①,其实这 7 个工作模式中,除了 user 和 system 外的其他 5 个工作模式都是异常模式(比如 IRQ 和 FIQ 模式是硬件中断,Supervisor 模式是软件中断,Abort

^① v6 体系架构后引入了更多的处理器模式。

和 Undefined 模式是异常)。这与 7 个工作模式相对应的, ARM 在硬件上设计了每个工作模式专用的部分通用寄存器堆, 比如 IRQ 状态下的 R13、R14、SPSR; FIQ 下的 R8~R14、SPSR。虽然这些寄存器拥有相同的名字, 但却是不同的硬件寄存器, 用户所访问的物理寄存器取决于当前 CPU 所处的工作模式, 比如在 IRQ 模式下访问的是 IRQ 模式专用的 R13 和 R14, 而在 USER 模式下访问的 R13 与 R14 则是属于该模式下另外的物理寄存器。ARM 这样设计是为了使处理器在进行状态切换时不需要进行访存操作或是减少访存操作 (主要是利用堆栈进行数据保存)。

5) 指令等长

ARM 的另外一个特点是不管在 ARM 状态还是 Thumb 状态所有的指令都是等长的, ARM 状态下是 32 位, Thumb 状态下是 16 位。这一点也是 ARM 作为 RISC 机器的一个非常重要的特点。指令等长的设计大大简化了 CPU 的取指逻辑和其他硬件逻辑。

当然, ARM 架构在遵循传统的 RISC 特征外, 还采用了一些特别的技术, 在保证高性能的前提下尽量缩小芯片的面积, 并降低功耗:

① 所有的 ARM 指令都可根据前面的执行结果决定是否被执行, 从而提高指令的执行效率。这也是 ARM 架构中非常有特色的条件执行技术, 一般 CPU 架构都支持条件跳转指令, 也就是是否跳转取决于前面指令的执行结果 (比如相等、大于、小于等)。ARM 将这个技术扩展到几乎所有指令, 当前指令是否真正被执行取决于前面的指令执行结果。这样做的好处有两个: 第一减少了不必要的跳转指令, 增加了指令密度; 第二, 因为减少跳转指令而由此减少指令流水线被打断的情况。当然, 任何事物都有利有弊, 当 ARM 架构发展到超标量乱序执行的 v7 架构时, 为了兼容条件执行技术, 处理器的架构会变得异常复杂。

② 可用加载/存储指令批量传输数据, 以提高数据的传输效率。ARM 的 LDM 和 STM 指令可以在一条指令中最多一次加载或存储 16 个 32 位数据, 虽然这些指令无法在一个周期内完成, 但由于降低了取指数量和充分利用外部 DDR 存储器的 Burst 特性, 总体上提升了处理器存取批量数据的能力。程序员可以利用这类指令高效地实现批量内存传输 (比如 memcpy) 和堆栈操作。

③ 可在一条数据处理指令中同时完成逻辑处理和移位处理。ARM 的数据处理指令通常采用三操作数模式, 即第一源操作数、第二源操作数和目的操作数。在处理器的微架构设计中, 第一源操作数直接由寄存器堆送往 ALU 运算单元, 而第二源操作数在由寄存器堆取出后, 要先经过一个桶形移位器移位后再送入 ALU。这样就可以在一条指令中完成移位与运算操作, 提高了指令效率。

④ 毫无疑问, Thumb 指令集是 ARM 系列处理器的另一个重要特征。与 ARM 32 位指令集不同, Thumb 指令集采用 16 位指令长度, 在牺牲一定性能的前提下, 可以使得代码密度提高 1/3, 这对于低成本系统 (比如外部存储器总线只有 16 位) 而言具有一定的优势。ARM 最早在 ARM7TDMI (v4T 架构) 处理器上实现 Thumb 指令集, ARM v6T2 架构进一步扩展了 Thumb 指令集的功能, 提出了所谓的 Thumb 2 技术 (注意, 是 Thumb 2 技术, 而不是 Thumb 2 指令集)。Thumb 2 技术解决了 Thumb 指令集的一些限制问题 (比如在中断处理程序中不能使用 Thumb 指令, Thumb 指令的跳转范围偏小等), 使得 16 位指令和 32 位指令可以混编。Thumb2 技术已成为 ARM v7 架构的标准配置。

⑤ 除了传统的 RISC 指令外, ARM 处理器从 v5EJS 开始支持多媒体 (数字信号处理) 扩展指令和 Jazelle 技术, 前者主要包括带饱和运算的乘加指令, 用于对媒体数据的处理; 后者则是 ARM 用于支持 Java 加速的硬件引擎。从 v6 体系架构开始, ARM 处理器增加了 SIMD 指令用于支持数据并发处理。在 v7 架构中, NEON 协处理器被集成在 ARM 主处理器中, 用于数据信号处理 (DSP) 和数据并发相关的计算任务。这些技术的集成大大加强了 ARM 处理器进行媒体数据运算的能力。

4.2.2 ARM7TDMI 编程模型

ARM7TDMI CPU 是 ARM 公司第一款非常成功的处理器内核，同时 ARM7 系列处理器的架构（v4T）基本奠定了后续所有处理器的基本结构，即使今天最先进的 v7 和 v8 架构的处理器也都是作为 v4T 架构的超集进行功能的扩展。另外，比较而言，ARM7 系列处理器的结构简单得多，非常适宜作为教学内容进行讲解。基于以上 3 点考虑，本书将通过 ARM7TDMI 来介绍 ARM 处理器的编程模型，而 ARM 处理器的最新进展以及相关编程模型将在 4.2.7 节和 4.2.8 节中介绍。

1. ARM7TDMI CPU 简介

ARM7 系列微处理器为低功耗的 32 位 RISC 处理器，最适合用于对价位和功耗要求较高的消费类应用。ARM7 微处理器系列具有如下特点：

- 具有嵌入式 ICE 逻辑（片上在线仿真），调试开发方便。
- 极低的功耗，适合对功耗要求较高的应用，如便携式产品。
- 能够提供 0.9MIPS/MHz 的三级流水线结构。
- 代码密度高并兼容 16 位的 Thumb 指令集。
- 对操作系统的支持广泛，包括 Windows CE、Linux、Palm OS 等。

ARM7 系列微处理器的主要应用领域为：工业控制、Internet 设备、网络和调制解调器设备、移动电话等多种多媒体和嵌入式应用。

ARM7 系列微处理器包括如下几种类型的核：ARM7TDMI、ARM7TDMI-S、ARM720T、ARM7EJ。其中，ARM7TMDI 是目前使用最广泛的 32 位嵌入式 RISC 处理器之一，其简单和全静态设计的特点尤其适合于对价格和功耗敏感的应用产品，广泛应用于各类消费、汽车电子、工业控制等产品中。ARM7TDMI 是在 ARM7 基础上的改进版本，除了有 ARM7 的 32 位集成芯片的基本特色外，它还有如下特点：Thumb16 位指令集、Debug 支持功能（使处理器能够暂停以响应一个 debug 请求）、Multiplier（增加了硬件乘法器，执行功能更强大，可以产生 64 位的数据结果）、嵌入式 ICE（支持指令断点和数据观察点）。这也是 ARM7TDMI 的名称由来。ARM7 系列微处理器的特性比较见表 4-5。

表 4-5 ARM7 系列微处理器的特性比较

ARM7 性能特征								
	Cache 大小 (指令/数据)	紧密耦合存 储器 (TCM)	存储器 管理	AHB 总线 接口	Thumb	DSP 指令	Jazelle	体系结 构版本
ARM7TDMI	无	无	无	有	有	无	无	v4T
ARM7TDMI-S	无	无	无	有	有	无	无	v4T
ARM7EJ-S	无	无	无	有	有	有	有	v5TEJ
ARM720T	8KB	无	MMU	有	有	无	无	v4T

2. ARM7TDMI 的组成

ARM7TDMI 处理器的主要组成部分如下：

- 寄存器堆（register bank）：用来存储处理器的状态。它有两个读端口和一个写端口，而 R15（程序寄存器）则有三个读端口和一个写端口。
- 桶形移位器（barrel shifter）：能使第二源操作数移动或循环任意位。
- ALU：完成指令集所需要的算术和逻辑运算功能。
- 地址寄存器与累加器：选择或使用已有的存储器地址，在需要时产生下一个地址。
- 数据寄存器（data register）：用来暂存输入或传出存储器的数据。

➤ 指令译码器和控制逻辑。

图 4-10 所示是 ARM7TDMI 的基本框图。

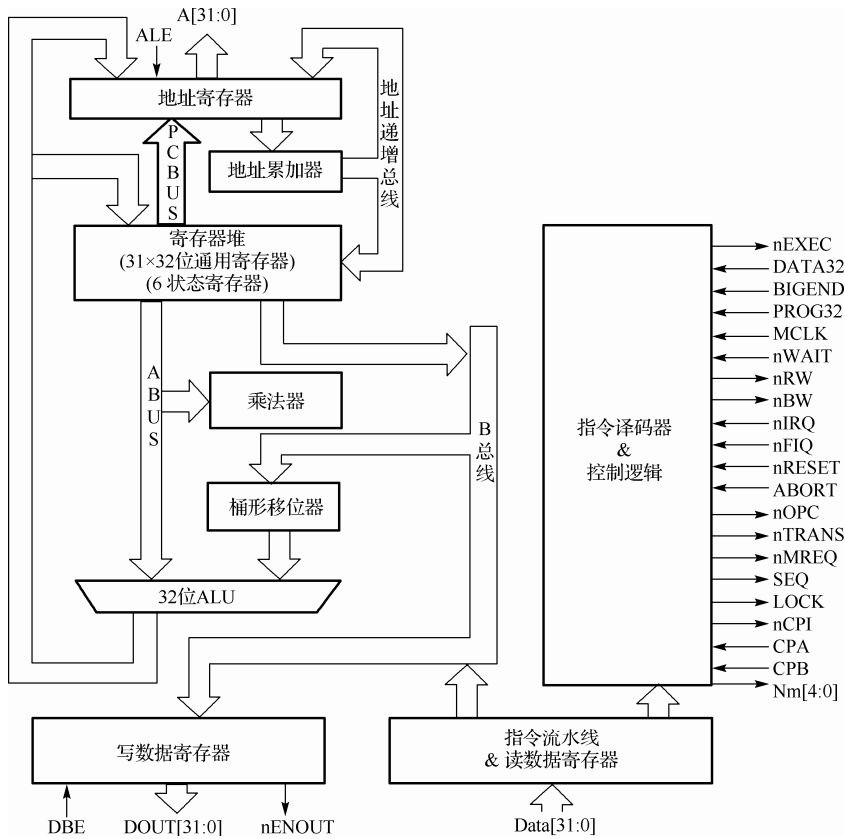


图 4-10 ARM7TDMI 框图

在一条数据处理指令中，将采用两个寄存器作为源操作数。B 总线上的数据（也就是指令中的第二个源操作数）将通过移位器移位后与 A 总线上的数据在 ALU 中进行计算，结果被写回寄存器堆。程序寄存器(PC, 也就是 r15 寄存器)中的值将存进地址寄存器, 该值再从这里放进地址加法器(Address Incrementer), 更新后的地址值再写回 R15 (PC) 和地址寄存器, 作为下一条指令的地址。

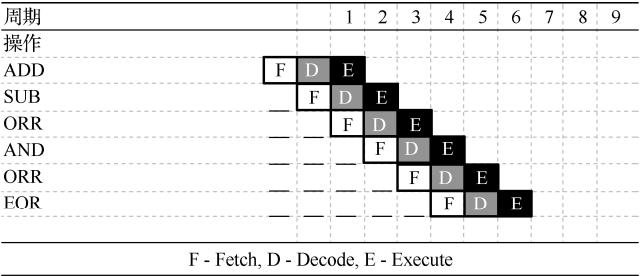
3. ARM7TDMI 的 3 级流水线

ARM7TDMI 的指令流水线采用经典的 3 级流水线:

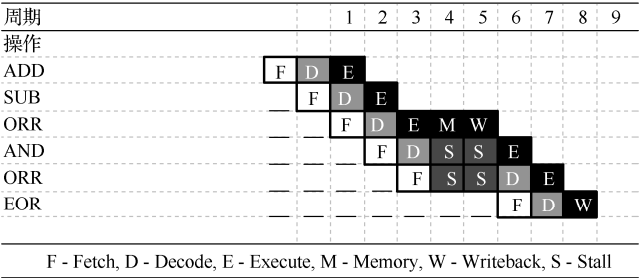
- 取指 (Fetch): 从存储器中取出指令并放进指令流。
- 译码 (Decode): 指令译码, 数据控制信号将准备下一个周期。这一阶段, 指令拥有译码逻辑, 但不拥有数据通路 (Data Path)。
- 执行 (Execute): 指令拥有数据通路; 寄存器组中的源操作数被读取, 第二源操作数被移位, ALU 负责将两个源操作数进行计算, 并将产生的结果写回目的寄存器。

图 4-11(a)所示是 ARM7TDMI 理想状况下的流水线, 图中第 1 个周期, ADD 指令处于执行状态, SUB 指令处于译码状态, 同时 CPU 正在将 ORR 指令进行取指。第 2 个周期, SUB 指令处于执行状态, ORR 指令处于译码状态, 并且开始取下一条 AND 指令。理想流水线的状况下, 任一周期中都有 3 条指令分别处于流水线的不同阶段 (可以理解为 CPU 同时在处理 3 条指令), 每一个周期结束时都会有

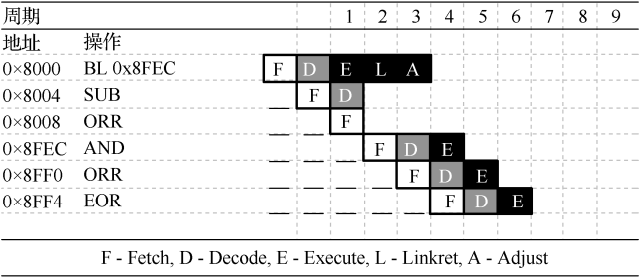
一条指令执行完毕。因此对于某一特定的指令而言，它的执行至少要经过 3 个周期，但是站在指令流的角度看，每个周期都会执行完毕一条指令，也就是平均而言每条指令的执行时间是一个时钟周期(CPI 等于 1)。需要说明的是，在具有流水线的处理器架构中，PC 当前值并不是指向正在被执行的指令地址，而是指向正在被取指的地址（如表 4-6 所示）。



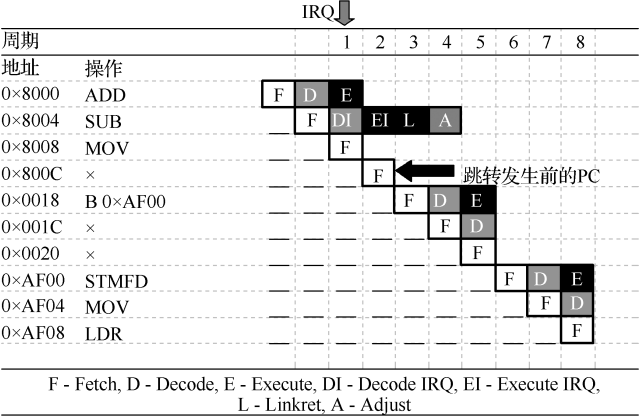
(a) 理想的指令流水线



(b) LDR 指令的流水线



(c) 跳转指令的流水线



(d) 中断发生时的指令流水线

图 4-11 ARM7TDMI 的 3 级流水线

表 4-6 ARM7TDMI 各个流水级所做的工作

ARM (指令的地址)	Thumb (指令的地址)	流水线执行阶段	所做的工作
PC	PC	取指	指令从存储器中取出
PC-4	PC-2	译码	对指令进行译码
PC-8	PC-4	执行	从寄存器组中读出寄存器的值，执行移位和 ALU 操作，结果写回寄存器组

注：PC 指向被取指的指令，而不是指向正在执行的指令。在正常操作过程中，在执行一条指令的同时对下一条指令进行译码，并将第三条指令从存储器中取出。

当然，CPU 不会总是处在理想流水线的状态，多数情况下流水线都会被打断，致使系统无法真正做到每个时钟周期完成一条指令执行的性能。图 4-11(b)所示是由于 LDR 指令而引起的流水线中断，在第 3 个周期，LDR 指令处于执行阶段，在这个周期中 CPU 的 ALU 计算出所要装载数据的地址。与此同时 CPU 在第 3 个周期还完成了 AND 指令的译码和 ORR 指令的预取。但是在第 4 个周期，CPU 会在流水线中插入一个 M（Memory）周期，用于访问第 3 周期所指定的内存地址，在周期结束前数据被读入到 CPU（这又是一个理想状况，通常只有在数据 Cache 命中的情况下这个数据装载才有可能在一个周期内完成，否则由于外部存储器往往要比 CPU 慢得多，为了等待数据的到来 CPU 可能会连续插入多个 M 周期）。在第 5 个周期，CPU 将获得的数据写回到指定的目的寄存器（这个写回的过程需要占用 ALU），我们将这个周期称为 W（Write Back）周期。在 M 周期和 W 周期，流水线的其他操作全部停顿，等待 W 周期结束后，在第 6 个周期才开始 AND 指令的执行。因此在这种情况下 CPU 需要用 6 个周期（最好情况，如果访存时间更长，则需要更长的时间）完成 4 条指令的执行，而此时的 CPI（Clock Per Instruction，每指令时钟数）就由理想的 1 变成了 1.5。

图 4-11(c)所示为由于跳转指令而引起的流水线停顿。第 1 个周期，BL 指令处于执行阶段，并通过 ALU 生成了目标地址，SUB 指令已被译码，ORR 指令已被预取。第 2 个周期，CPU 开始预取处于目标地址的 AND 指令，并同时跳转发生前的 PC 地址（0x8008）保存到 LR 寄存器（R14），因此我们把这个周期也称为 L（Link）周期。请注意被保存在 R14 中的 0x8008 并不是程序的返回地址，这个地址比所要返回的地址要大 0x4，因此在第 3 个周期，CPU 要修正 R14 中的值，将其减去 0x4，这样 R14 中的地址就变成了 0x8004，而这个地址才是从函数调用返回后应该执行的第一条指令，由于这个调整的过程，这个周期也被称为 A（Adjust）周期。直到第 4 个周期，跳转目的地址（0x8FEC）的指令才会进入执行阶段，因此跳转指令的执行至少需要 3 个周期。

图 4-11(d)所示为中断引起的流水线中断。第 1 个周期，外部中断 IRQ 向 CPU 提起中断请求，但 CPU 只有在本周期内的指令执行完毕后才会响应这个中断请求。第 2 个周期，系统开始响应中断，切换到 ARM 状态（32 位指令），获得中断向量地址，将 CPSR 保存到相应的 SPSR，为 CPSR 置位等（关于 ARM 处理器的中断响应过程参见 4.2.5 节）。第 3 个周期，与跳转指令类似，CPU 进入 L 周期，将 PC 保存到 R14 寄存器，并且开始取中断向量表中的指令；第 4 周期对 R14 的值进行调整（减 4），也就是 A 周期，同时对中断向量表中的指令开始译码。第 5 个周期，中断向量表中的跳转指令被执行，并在第 6 个周期开始取中断处理程序的第一条指令，该指令进入执行状态是第 8 个周期。因此中断的响应时间至少是 7 个周期。

虽然 ARM7TDMI 处理器的三级流水线在硬件实现上比较简单，但从每一级流水线所承担的工作内容来看，显然第三级（执行级）所负担的工作非常多，至少包括读源寄存器、第二源操作数移位、ALU 运算，以及将运算结果写回到寄存器堆的任务。由于执行的任务比较多，因此第三级运算所需要的时间更多，这就决定了处理器的频率难以提高（因为主频的倒数就是周期长度）。为了进一步提升处理器的主频，从 ARM9 开始，采用更多级数的流水线设计。比如，ARM9 采用

5 级流水，ARM11 采用 8 级流水，而 Cortex A9 采用 11 级流水。图 4-12 对比了 ARM7 的 3 级流水线与 ARM9 的 5 级流水线。ARM9 采用的 5 级流水线可以说是非常经典的流水线设计，很多早期的 RISC 处理器都采用这样的设计架构（比如早期的 MIPS 架构）。在这个设计中整个执行阶段被分解为执行阶段(E)、访存阶段(M)和写回阶段(W)。对于没有访存需求的指令（除了 LDR/STR 外的其他指令），M 周期被轮空，而将 ALU 运算结果写回到寄存器堆的操作被分配到了写回阶段（W 周期）。

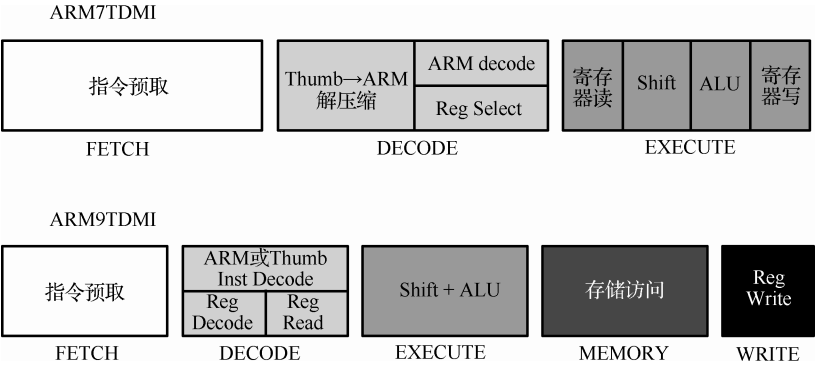


图 4-12 ARM7 的 3 级流水线与 ARM9 的 5 级流水线

由于采用了更多的流水线级数，ARM9 处理器在同样制造工艺下的主频比 ARM7 处理器有了较大的提升。当然，并不是流水线级数越多越好，随着流水线级数的增加，由于跳转指令、中断等原因引起的流水线停顿，以及由于指令之间的相关性而引起的冒险都会使流水线的性能降低。由此而引入的指令分支预测、乱序执行等技术虽然在很大程度上解决了这些问题，但所带来的硬件复杂性以及功耗也是非常可观的。处理器设计者需要在这些开销与性能收益之间小心地进行折中设计。另外需要说明的是，虽然在后续的架构中 ARM 引入了更多的流水线级数，但为了保持指令集的兼容性，所有涉及 PC 与当前被执行指令地址之间偏移量的关系都被固定下来保持不变，程序员不需要关心在不同架构间可能存在的不兼容。

4. ARM7TDMI 的 7 种工作模式

ARM7TDMI 支持的运行模式有 7 种，它们分别为：

- 用户模式（usr）：用于运行用户应用程序的非特权模式。
- 快速中断模式（fiq）：用于快速中断处理。
- 外部中断模式（irq）：用于通用的硬件中断处理。
- 管理模式（svc）：操作系统使用的保护模式。
- 数据访问终止模式（abt）：当数据或指令预取终止时进入该模式，可用于虚拟存储及存储保护。
- 系统模式（sys）：运行具有特权的操作系统任务。
- 未定义指令中止模式（und）：当未定义的指令执行时进入该模式，可用于支持硬件协处理器的软件仿真。

大多数的应用程序运行在用户模式下，当处理器运行在用户模式下时，某些被保护的系统资源是不能被访问的。比如，在用户模式下，程序只能读取 CPSR（当前程序状态寄存器）中的特定位（比如是否进位？是否为零？是否溢出？是否为负？），但是程序员无法通过指令显式地修改程序状态字（对于 CPSR 只能读不能写）。另外，在用户模式下，程序员也无法访问协处理器指令。

除用户模式以外，其余的所有 6 种模式称为非用户模式，或特权模式（Privileged Modes）。其中，

除去系统模式以外的 5 种又称为异常模式（Exception Modes），用于处理中断或异常，以及需要访问受保护的系统资源等情况。特权模式下，程序可以读取或修改 CPSR 的所有内容，也可以执行所有的协处理器指令。

系统模式属于特权模式，可以访问所使用的系统资源，也可以直接进行处理器的模式切换。它主要供操作系统任务使用。

ARM 微处理器的运行模式可以在特权模式下通过修改 CPSR 的特定位而进行切换，比如从 FIQ 模式切换到 IRQ 模式，或者从 SVC 模式切换到 ABT 模式。由于在用户模式（非特权模式）下无法修改 CPSR，因此一旦 CPU 进入该模式，回到特权模式的唯一办法就是通过外部中断、异常或者执行软件中断指令（通常操作系统实现系统调用就是通过执行软中断指令进入内核态的）来实现模式切换。ARM 处理器复位后进入的工作模式是 SVC 模式。

在 ARM v6 架构中，为了支持更好的安全性，引入了 Trust Zone 技术。Trust Zone 技术将系统分为安全态（Secure State）与普通态（Non-Secure State），运行在普通态下的软件无法访问安全态下的内存，为了连接这两个状态，v6 架构引入了监控（Monitor）工作模式。为了更好地支持虚拟化运行（一种可以同时支持不同操作系统的技术），从 v7 架构中开始支持超管理（Hypervisor）模式。因此，在 v7 架构的处理器中一共支持 9 种工作模式。

5. ARM7TDMI 的寄存器

ARM v6 架构以前的处理器有 37 个物理寄存器，其中包括 31 个通用 32 位寄存器、1 个当前程序状态寄存器（CPSR）和 5 个保存程序状态寄存器（SPSRs）。这些寄存器都是 32 位寄存器。

ARM v6 架构前的处理器有 7 种不同的工作模式，在每一种处理器工作模式中有一组相应的寄存器组。一个寄存器组包括 16 个通用寄存器（R0~R15，其中 r15 是程序计数器 PC）、一个或两个状态寄存器。在所有的寄存器中，寄存器被安排成部分重叠的组（overlapping bank）。有些寄存器是各模式共用同一个物理寄存器，有些寄存器是各模式自己拥有独立的物理寄存器。当处于不同的模式时，某些 ARM 寄存器可能对应不同的物理寄存器，如寄存器 SPSR，每种处理器模式下都对应一个独立的物理寄存器（一个例外是：用户模式和系统模式采用完全相同的物理寄存器，并且这两个模式下也没有 SPSR 与之对应）。图 4-13 列出了 ARM 的 37 个物理寄存器，图中阴影部分的寄存器即为相应处理器模式的影子寄存器。影子寄存器只有特定的处理器模式才可以访问。注意，这些影子寄存器或者分组寄存器，虽然拥有相同的名字，比如在汇编代码中都使用 r13，但是在不同模式下访问的物理寄存器其实是完全不同的寄存器。为了区分，在图 4-13 中我们将其表示为 r13_fiq、r13_irq、r13_svc、r13_abt、r13_und，因此实际 CPU 中共有 6 个同名的 r13（还有一个是用户模式下的 r13），在某个特定模式下只能访问属于这个模式的 r13。
















在最新的 v7 架构中，基本的编程模型没有改变，只是增加了两个新的工作模式下对应的分组寄存器。比如，Monitor 模式下的 r13_mon、r14_mon、SPSR_mon，Hypervisor 模式下的 r13_hyp、SPSR_hyp。Hypervisor 模式与 User 模式共用 r14 寄存器，并增加了一个叫作 ELR_hyp 的寄存器。

通用寄存器包括 R0~R15，可以分为未分组寄存器和分组寄存器。
















1) 未分组寄存器 R0~R7

在所有的处理器工作模式下，未分组寄存器都指向同一个物理寄存器，它们是真正的通用寄存器，没有体系结构所隐含的特殊用途。因此，在中断或异常处理进行处理器工作模式转换时，由于不同的处理器工作模式均使用相同的物理寄存器，可能会造成寄存器中数据的破坏，这一点在进行程序设计时应引起注意。

ARM状态下的通用寄存器与程序计数器

System & User	FIQ	Supervisor	Abort	IRQ	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	 R8_fiq	R8	R8	R8	R8
R9	 R9_fiq	R9	R9	R9	R9
R10	 R10_fiq	R10	R10	R10	R10
R11	 R11_fiq	R11	R11	R11	R11
R12	 R12_fiq	R12	R12	R12	R12
R13	 R13_fiq	 R13_svc	 R13_abt	 R13_irq	 R13_und
R14	 R14_fiq	 R14_svc	 R14_abt	 R14_irq	 R14_und
R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)

ARM状态下的程序状态寄存器

CPSR	<table><tr><td>CPSR</td></tr><tr><td> SPSR_fiq</td></tr></table>	CPSR	 SPSR_fiq	<table><tr><td>CPSR</td></tr><tr><td> SPSR_svc</td></tr></table>	CPSR	 SPSR_svc	<table><tr><td>CPSR</td></tr><tr><td> SPSR_abt</td></tr></table>	CPSR	 SPSR_abt	<table><tr><td>CPSR</td></tr><tr><td> SPSR_irq</td></tr></table>	CPSR	 SPSR_irq	<table><tr><td>CPSR</td></tr><tr><td> SPSR_und</td></tr></table>	CPSR	 SPSR_und
CPSR															
 SPSR_fiq															
CPSR															
 SPSR_svc															
CPSR															
 SPSR_abt															
CPSR															
 SPSR_irq															
CPSR															
 SPSR_und															


 = 分组寄存器

图 4-13 ARM v6 架构前的寄存器分组

2) 分组寄存器 R8~R14

对于分组寄存器，它们每一次所访问的物理寄存器与处理器当前的工作模式有关。

对于 R8~R12 来说，每个寄存器对应两个不同的物理寄存器，一组为 FIQ 模式，另一组为除 FIQ 模式以外的其他模式。当使用 FIQ 模式时，访问寄存器 R8_fiq~R12_fiq；当使用除 FIQ 模式以外的其他模式时，访问寄存器 R8~R12。FIQ 模式下的 R8_fiq~R12_fiq 这几个影子寄存器的设计是为了进行快速中断处理，在 FIQ 中断处理程序中如果仅使用 R8_fiq~R12_fiq 而不使用其他的寄存器，可以不用在堆栈中保存任何寄存器，从而加快了 FIQ 中断处理速度。

对于 R13、R14 来说，每个寄存器对应 6 个不同的物理寄存器（v6 架构前），其中的一个是用户模式与系统模式共用，另外 5 个物理寄存器对应于其他 5 种不同的运行模式。即：

```
R13_<mode>
R14_<mode>
```

其中，mode 为以下几种模式之一：usr、fiq、irq、svc、abt、und。在这里使用伪代码来表示 R13 和 R14 的影子寄存器，在实际编程时，如果在指令中访问 R13（指令中无需写成 R13_<mode> 的形式，只需记作 r13），处理器会根据当前的工作模式来判断当前访问的是哪个物理寄存器。如果当前处理器的工作模式为 svc，则对 R13_<svc> 进行操作。在本书中记作 R13_<mode> 是为了方便读者理解。

寄存器 R13 在 ARM 指令中常用作堆栈指针，称为 SP。但这只是一种习惯用法，用户也可使用其他的寄存器作为堆栈指针。而在 Thumb 指令集中，某些指令强制性地要求使用 R13 作为堆栈指针。由于处理器的每种运行模式均有自己独立的物理寄存器 R13，在用户应用程序的初始化部分，一般都要初始化每种模式下的 R13，使其指向该运行模式的栈空间。这样，当程序的运行进入异常模式时，可以将需要保护的寄存器值放入 R13 所指向的堆栈，而当程序从异常模式返回时，则从对应的堆栈中恢复，采用这种方式可以保证异常发生后程序的正常执行。

R14 也称作子程序连接寄存器（Subroutine Link Register）或连接寄存器 LR。当执行 BL 程序调用指令时，硬件会自动将返回地址（实际上是程序计数器 pc-4）保存到该寄存器。其他情况下，R14 用作通用寄存器。与之类似，当发生中断或异常时，对应的分组寄存器 R14_svc、R14_irq、R14_fiq、R14_abt 和 R14_und 用来保存返回地址（实际上是程序计数器 pc-4）。

User、IQR、Supervisor、Abort 和 Undefined 模式的每一种都包含两个分组寄存器 R13 和 R14 的映射，允许每种模式都有自己的堆栈指针寄存器和连接寄存器。

6. 程序计数器 PC（R15）

寄存器 R15 用作程序计数器（PC）。在 ARM 状态下，位[1:0]为 0，位[31:2]用于保存 PC；在 Thumb 状态下，位[0]为 0，位[31:1]用于保存 PC。虽然可以用作通用寄存器，但是有一些指令在使用 R15 时有一些特殊限制，若不注意，执行的结果将是不可预料的。R15 虽然也可用作通用寄存器，但一般不这样使用。R15 通常用于某些特殊场合：

- （1）读程序计数器。一般情况下指令读出的 R15 的值是当前被执行指令的地址加上 8 个字节。读 PC 主要用于快速地对临近的指令和数据进行位置无关寻址。
- （2）写程序计数器。写 R15 的通常结果是将写到 R15 中的值作为指令地址，并以此地址发生转移。

7. 程序状态寄存器

CPSR（Current Program Status Register，当前程序状态寄存器）可在任何运行模式下被访问，它包括条件标志位、中断禁止位、当前处理器模式标志位，以及其他一些相关的控制和状态位。

在 fiq、irq、svc、abt、und 等 5 种处理器工作模式下又都有一个专用的寄存器：SPSR（Saved Program Status Register，备份的程序状态寄存器），当异常发生时，SPSR 用于保存 CPSR 的当前值，从异常退出时则可由 SPSR 来恢复 CPSR。注意，这一点和传统处理器在响应中断时自动将程序状态字压栈不同，ARM 是 Load/Store 架构，除了执行 Load 和 Store 两类指令外，CPU 不访存（当然，取指除外）。因此，ARM 通过 LR 寄存器保存返回地址，SPSR 保存程序状态字。如果程序员需要将这些值压入堆栈，则需要显式地调用 Store 指令进行入栈操作。

由于用户模式和系统模式不属于异常模式，它们没有 SPSR，在这两种模式下访问 SPSR 时结果是未知的。

CPSR 在用户编程时用于存储条件码。例如，这些位可用来记录比较操作的结果和控制条件转移是否发生。用户级程序员通常不关心该寄存器是如何配置的。ARM CPSR 寄存器的格式如图 4-14 所示。

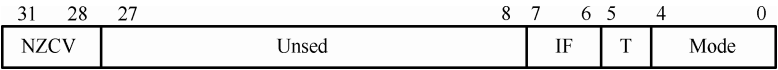


图 4-14 ARM CPSR 寄存器的格式（v4T 架构）

寄存器的低位用于控制处理器的模式（由 Mode 位来控制）、处理器的工作状态是 ARM 状态还是 Thumb 状态（由 T 位来控制）、中断使能（由 I 和 F 位来控制），而且被保护以防止用户级程序改变它们。条件码标志位位于寄存器的高 4 位。

- mode 位的意义见表 4-7。
- T 位的意义如下：
- T=0 表示执行 ARM 指令。
 - T=1 表示执行 Thumb 指令。

表 4-7 mode 位的意义

M[4:0]	处理器模式
0b10000	User
0b10001	FIQ
0b10010	IRQ
0b10011	Supervisor
0b10111	Abort
0b11011	Undefined
0b11111	System

中断禁止位的意义如下：

- 当 I=1 时禁止 IRQ 中断。
- 当 F=1 时禁止 FIQ 中断。

条件标志位的意义如下：

- N：负数。改变标志位的最后的 ALU 操作产生负数结果（32 位结果的最高位是 1）
- Z：零。改变标志位的最后的 ALU 操作产生 0 结果（32 位结果的每一位都是 0）。
- C：进位。改变标志位的最后的 ALU 操作产生到符号位的进位。
- V：溢出。改变标志位的最后的 ALU 操作产生到符号位的溢出。

注意，尽管以上 C 和 V 的定义看起来颇为复杂，但使用时不需要详细了解它们的操作。在大多数情况下有一个简单的条件测试，不需要程序员计算出条件码的精确值即可得到需要的结果。

在 ARM 状态下，任一时刻可以访问以上所讨论的 16 个通用寄存器和一到两个状态寄存器。在非用户模式（特权模式）下，则可访问到特定模式的分组寄存器。

ARM v7 架构对 CPSR 进行了较大的扩展，如图 4-15 所示。我们仅简单介绍一下与 v4T 架构不同的部分。

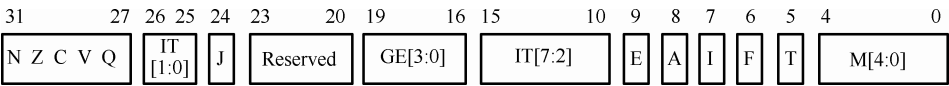


图 4-15 ARM v7 架构的 CPSR 寄存器

Q：用于表示饱和乘加时的饱和状态。

IT[7:0]：这 8 位被分在两个不同的地方存储，用于 Thumb2 指令的 If-Then 条件执行。

J：用于表示当前 CPU 处于 Jazelle 状态。

GE[3:0]：用于某些 SIMD 指令。

E：用于控制 LDR/STR 指令的访存印第安序。

A：用于关闭异步终止（Asynchronous Abort）。

8. ARM7TDMI 的堆栈

堆栈有软件堆栈和硬件堆栈之分。软件堆栈是指堆栈的管理完全由软件进行，硬件堆栈是指由硬件来管理堆栈。ARM 的堆栈是软件堆栈，完全由软件来管理。

ARM 的 7 种工作模式分别有 6 个堆栈指针寄存器 r13（v6 架构前），记作 sp_usr、sp_svc、sp_fiq、sp_irq、sp_abt、sp_und，分别对应于用户/系统模式、管理模式、FIQ 模式、IRQ 模式、中止模式、未定义模式下的堆栈指针寄存器。在系统初始化时，要由软件来给这 6 个堆栈指针寄存器赋一个初值，指向各种处理器模式下的堆栈的栈顶。这样，各个处理器模式就拥有了各自的堆栈空间。

ARM 处理器中的堆栈同其他处理器一样，也是按先进后出的方式工作的，使用一个称作堆栈指针的专用寄存器（即 r13，也记作 sp）指示当前的操作位置，堆栈指针总是指向栈顶。当堆栈指针指向最后压入堆栈的数据时，称为满堆栈（Full Stack）；而当堆栈指针指向下一个将要放入数据的空位置时，称为空堆栈（Empty Stack）。同时，根据堆栈的生成方式，又可以分为递增堆栈（Ascending Stack）和递减堆栈（Decending Stack）；当堆栈由低地址向高地址生成时，称为递增堆栈；当堆栈由高地址向低地址生成时，称为递减堆栈。这样就有 4 种类型的堆栈工作方式，ARM 微处理器支持这 4 种类型的堆栈工作方式。

- 满递增堆栈：堆栈指针指向最后压入的数据，且由低地址向高地址生成。
- 满递减堆栈：堆栈指针指向最后压入的数据，且由高地址向低地址生成。
- 空递增堆栈：堆栈指针指向下一个将要放入数据的空位置，且由低地址向高地址生成。
- 空递减堆栈：堆栈指针指向下一个将要放入数据的空位置，且由高地址向低地址生成。

堆栈的入栈操作和出栈操作要由软件来进行，ARM 提供对堆栈进行操作的指令 LDMFD 和 STMFD。这两条指令符合 ARM 的 Load/Store 指令架构。FD 是指满递减堆栈。

堆栈用于中断和操作系统中任务调度时的上下文保护，保存 ARM 的寄存器、传递参数、保存函数中的局部变量等。当然，参数和局部变量也可以用 ARM 的寄存器来传递，当寄存器不够用时使用堆栈来传参和保存局部变量，这些规则在 4.2.6 节有详细介绍。

9. ARM7TDMI 的存储系统

除了处理器的寄存器状态外，ARM 系统还有存储器状态。存储器可以看作是序号为 $0 \sim 2^{32}-1$ 的线性字节阵列。数据项可以是 8 位字节、16 位半字和 32 位字。字总是以 4 字节的边界对准，半字则以偶字节的边界对准。

存储器组织如图 4-16 所示。图中画出了存储器的一个小区域，其中每一个字节都有唯一的号码。字节可以占有任一位置，图中给出了几个例子。长度为 1 个字的数据项占有一组 4 字节的位置，该位置开始于 4 的倍数的字节地址。

图 4-16 中包含了两个这样的例子。半字节占有两个字节位置，该位置开始于偶数字节地址。

23	22	21	20
19	18	17	16
字8			
15	14	13	12
半字14		半字12	
11	10	9	8
字8			
7	6	5	4
字节6		半字4	
3	2	1	0
字节3	字节2	字节1	字节0

图 4-16 ARM 的存储器组织

这是标准的“小端”（Little Endian）格式，即 ARM 使用的存储器组织。新的 ARM 架构也可以配置为按“大端”格式访问存储器。下面简单介绍一下 ARM 体系结构的存储器格式。

ARM 体系结构将存储器看作是从零地址开始的字节的线性组合。从第 0 个字节到第 3 个字节放置第一个存储的字数据，从第 4 个字节到第 7 个字节放置第二个存储的字数据，依次排列。作为 32 位的微处理器，ARM 体系结构所支持的最大寻址空间为 4GB（ 2^{32} 字节）。

ARM 体系结构可以用两种方法存储字数据，称为大端格式和小端格式，具体说明如下。

1) 大端格式

在这种格式中，字数据的高字节存储在低地址中，而字数据的低字节则存放在高地址中，如图 4-17 所示。因此在大端格式中，图中 0x0 地址存放的 32 位数据是 0x3210；0x4 地址中的数据是 0x7654。

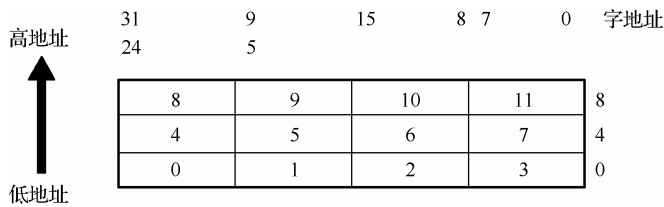


图 4-17 以大端格式存储字数据

2) 小端格式

与大端存储格式相反，在小端存储格式中，低地址中存放的是字数据的低字节，高地址存放的是字数据的高字节。同样是 0x3210 和 0x7654 这两个 32 位数存放在小端格式时，在内存中的排布就变成了如图 4-18 所示。

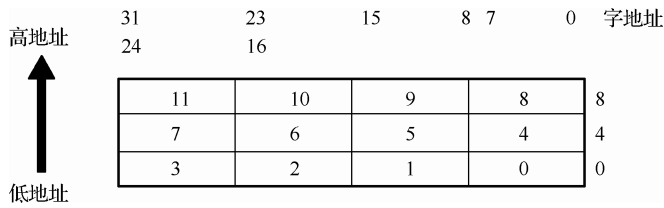


图 4-18 以小端格式存储字数据

4.2.3 ARM7TDMI 的指令集

1. ARM 指令集特点

对熟悉现代 RISC 指令集的读者来说，ARM 指令集看起来比传统的 RISC 处理器有更多的指令格式。事实也确实如此，虽然这使得指令译码更复杂，但同时也带来了较高的代码密度（相比传统的 RISC 架构而言）。大多数 ARM 处理器都应用于小型的嵌入式系统。对于这些系统，代码密度的优势超过了因译码复杂而导致的损失。Thumb 代码扩展了这一优势，使 ARM 比大多数 RISC 处理器有更好的代码密度。

典型的 CISC 处理器允许在很多指令中将某个存储器地址中的值作为一个源操作数或者目的操作数。同大多数 RISC 处理器一样，ARM 使用 Load-Store 体系结构。这就意味着指令集仅能处理（指加、减等）寄存器中（或指令中直接指定）的值，而且总是将这些处理的结果放回到寄存器中。针对存储器状态的唯一操作是将存储器的值复制到寄存器（Load 指令）或将寄存器中的值复制到存储器（Store 指令）。因此，所有的 ARM 指令都属于下列 3 种类型之一。

- 数据处理指令。这类指令只能使用和改变寄存器中的值。例如，一条指令能使两个寄存器相加，并将结果放到一个寄存器中。
- 数据传送指令。这类指令把存储器中的值复制到存储器中（Store 指令）或反之（Load 指令）。另外一种形式仅在系统代码中 useful，即交换存储器和寄存器的值。
- 控制流指令。一般指令在执行时使用存储于连续的存储器地址中的指令。控制流指令使执行切换到不同的地址。切换或者是永久的（转移指令），或者保存返回地址以恢复原来的执行顺序（函数调用指令），或者陷入系统代码（系统调用）。

ARM 指令集的主要特点包括：

- Load-Store 体系结构。

- 3 地址的数据处理指令（即两个源操作数寄存器和结果寄存器都独立设定）。
- 每条指令都可以根据上一条指令执行的结果来条件执行。
- 包含非常强大的多寄存器 Load 和 Store 指令，支持内存块的复制、批量数据的入栈出栈等操作。
- 能用在单时钟周期内执行的单条指令来完成一项普通的移位操作和一项普通的 ALU 操作。
- 通过协处理器指令集来扩展 ARM 指令集，包括在编程模式中增加了新的寄存器和数据类型。
- 在 Thumb 体系结构中高密度 16 位压缩形式表示的指令集。

2. ARM 指令的分类

ARM 微处理器的指令集是加载/存储型的，即指令集仅能处理寄存器中的数据，而且处理结果都要放回寄存器中，而对系统存储器的访问则需要通过专门的加载/存储指令来完成。

ARM 微处理器的指令集可以分为数据处理指令、加载/存储指令、跳转指令、程序状态寄存器（PSR）处理指令、协处理器指令和异常产生指令六大类，具体的指令及功能如表 4-8 所示（表中指令为基本 ARM 指令，不包括派生的 ARM 指令）。其中数据处理类指令占到了绝大多数，我们进一步将数据处理类指令又分为加减运算指令、乘法指令、逻辑运算指令、比较指令和数据传送指令。

表 4-8 ARM 指令分类及功能描述（V4T 架构）

助记符	指令功能描述	指令类别	
ADC	带进位加法指令	加减运算指令	数据处理指令
ADD	加法指令		
SUB	减法指令		
SBC	带借位减法指令		
RSB	逆向减法指令		
RSC	带借位的逆向减法指令		
MUL	32 位乘法指令	乘法指令	
MLA	乘加运算指令		
MVN	数据取反传送指令		
AND	逻辑与指令	逻辑运算指令	
ORR	逻辑或指令		
EOR	异或指令		
BIC	位清除指令		
CMN	比较反值指令	比较指令	
CMP	比较指令		
TEQ	相等测试指令		
TST	位测试指令		
MOV	数据传送指令	数据传送指令	
LDM	加载多个寄存器指令	LDR/STR 类指令	
LDR	存储器到寄存器的数据传输指令		
STM	批量内存字写入指令		
STR	寄存器到存储器的数据传输指令		
SWP	交换指令		
B	跳转指令	跳转指令	
BL	带返回的跳转指令		
BLX	带返回和状态切换的跳转指令		
BX	带状态切换的跳转指令		

续表

助记符	指令功能描述	指令类别
MRS	传送 CPSR 或 SPSR 的内容到通用寄存器指令	程序状态字指令
MSR	传送通用寄存器到 CPSR 或 SPSR 的指令	
CDP	协处理器数据操作指令	协处理器指令
MCR	从 ARM 寄存器到协处理器寄存器的数据传输指令	
MRC	从协处理器寄存器到 ARM 寄存器的数据传输指令	
LDC	存储器到协处理器的数据传输指令	
STC	协处理器寄存器写入存储器指令	
SWI	软件中断指令	
		异常产生指令

3. 指令的条件域

当处理器工作在 ARM 状态时，几乎所有的指令均可以根据 CPSR 中条件码的状态和指令的条件域有条件地执行。当指令的执行条件满足时，指令被执行，否则指令被忽略。

每一条 ARM 指令编码中都包含 4 位的条件码，位于指令的最高 4 位[31:28]。条件码共有 16 种组合。在助记符中，每种条件码可用两个字母表示，这两个字母可以添加在基本指令助记符的后面从而派生出新的指令。例如，跳转指令 B 可以加上后缀 EQ 变为 BEQ 表示“相等则跳转”，即当 CPSR 中的 Z 标志置位时发生跳转。

在 16 种条件标志码中只有 15 种可以使用，如表 4-9 所示，第 16 种（1111）为系统保留，暂时不能使用。需要说明的是条件码后缀 AL 表示 Always，也就是该指令总是被执行（无条件执行），每条指令缺省状况下都是无条件执行的，因此助记符中一般不写 AL。比如，助记符 ADDAL 和 ADD 是同样的含义，都表示无条件执行加法指令。

表 4-9 指令的条件码

条件码	助记符后缀	标志	含义
0000	EQ	Z 置位	相等
0001	NE	Z 清零	不相等
0010	CS	C 置位	无符号数大于或等于
0011	CC	C 清零	无符号数小于
0100	MI	N 置位	负数
0101	PL	N 清零	正数或零
0110	VS	V 置位	溢出
0111	VC	V 清零	未溢出
1000	HI	C 置位 Z 清零	无符号数大于
1001	LS	C 清零 Z 置位	无符号数小于或等于
1010	GE	N 等于 V	带符号数大于或等于
1011	LT	N 不等于 V	带符号数小于
1100	GT	Z 清零且（N 等于 V）	带符号数大于
1101	LE	Z 置位或（N 不等于 V）	带符号数小于或等于
1110	AL	忽略	无条件执行

正如前面所分析的，采用条件码一方面可以增加代码密度，另一方面由于减少了跳转指令而使得处理器的流水线尽可能不被打断而提升了执行性能。下面的例子给出了采用条件执行和不采用条件执行的区别。虽然在这个例子中采用条件执行只比非条件执行节省了 2~3 个周期的执行时间，但如果这段代码是在一个大循环中执行的（循环次数 10 000 次），那么总的节省下的执行时间可能达到 2 万~

3 万个周期，这时的优化效果就比较可观了。通常情况下程序员并不需要自己编写条件执行的机器代码，C 编译器会自动将代码优化为条件执行代码。

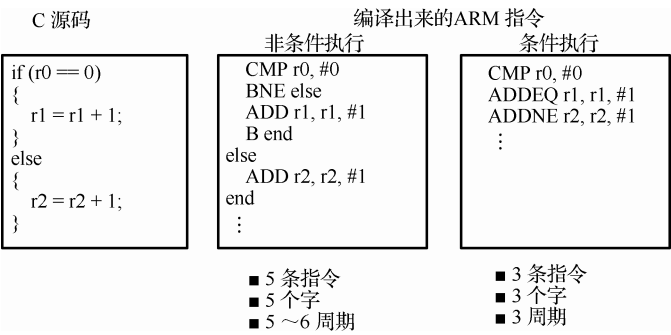


图 4-19 条件执行与非条件执行的对比

4. ARM 指令的一般编码格式

ARM 指令字长为固定的 32 位。一条典型的 ARM 指令编码格式如图 4-20 所示（这是数据处理指令的一般格式，对于 LDR/STR 指令和跳转指令及其他类型的指令，ARM 采用不同的编码格式）。

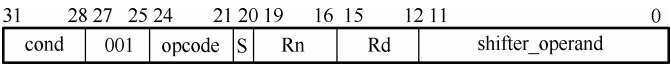


图 4-20 数据处理指令的编码格式

其中：

- Opcode 指令操作符编码。
- Cond 指令执行的条件编码。
- S 决定指令的操作是否影响 CPSR 的值（如果目的操作数 Rd 为 r15 且 S 置位，该指令还把 SPSR 的值恢复到 CPSR）。
- Rd 目标寄存器编码。
- Rn 第一源操作数的寄存器编码。
- Shifter_operand 表示第二源操作数（可能是立即数，也可能是寄存器加移位操作）。

一条典型的 ARM 指令助记符的语法格式如下：

```
<opcode>{<cond>}{S} <Rd>, <Rn>{, <operand2>}
```

虽然基本类型的 ARM 指令（由 opcode 决定）并不是很多，但是加上条件执行码（cond 位）和是否影响 CPSR（S 位）以及第二源操作数的变化就可以派生出很多相关指令。这也是 ARM 指令集的一个很重要的特点，初学者只需要记住指令基本助记符、条件码、S 位、第二源操作数的变化形式，就可以灵活变化出各种不同的指令。下面的例子给出了 ADD 指令的一些变化。

ADD r1, r2, r3	;最基本的 ADD 指令, r2+r3->r1
ADDEQ r1, r2, r3	;如果 CPSR 中的 Z 置位（相等），则执行 ADD
ADDNES r1, r2, r3	;Z 清零（不相等），则执行，并影响 CPSR 相应位
ADDGE r1, r2, r3 LSL #2	;GE 则执行, r3 内容逻辑左移 2 位后再与 r2 内容相加
ADDS r1, r2, r3 LSR r4	;r3 内容逻辑右移 r4 位后与 r2 相加, 结果写回 r1, 影响 CPSR
ADD r1, r2, #0x12	;r2 内容与立即数 0x12 相加, 结果写回 r1

5. ARM v4 架构指令集

下面按照 ARM 指令集的九大类指令：数据处理指令、第二操作数与移位、乘法指令与乘加指令、加载/存储指令、批量数据加载存储指令、数据交换指令、跳转指令、程序状态寄存器（PSR）处理指令、协处理器指令和异常产生指令分别进行描述。

1) 数据处理指令

数据处理指令可分为数据传送指令、算术逻辑运算指令和比较指令等。

数据传送指令用于在寄存器和存储器之间进行数据的双向传输。

算术逻辑运算指令完成常用的算术与逻辑的运算，该类指令不但将运算结果保存在目的寄存器中，同时更新 CPSR 中的相应条件标志位（需要指令中 S 位置位）。注意，对于数据处理指令，如果指令中 S 置位，并且目的寄存器为 r15（PC），则该指令除了将结果写回 PC 外，还将当前模式的 SPSR 恢复到 CPSR 中（该指令的运算结果并不影响 CPSR 的标志位）。在实际应用中，程序员通常利用这类指令实现异常（中断）处理程序的返回。

比较指令不保存运算结果，只更新 CPSR 中相应的条件标志位（这类指令缺省就影响 CPSR 标志位，不需要指令中指明 S 位）。

数据处理指令见表 4-10。

表 4-10 数据处理指令

操作码	操作数	描述	功能
算术运算			
ADC	Rd, Rn, Op2	带进位加法	$Rd=Rn+Op2+C$
ADD	Rd, Rn, Op2	加法	$Rd=Rn+Op2$
MOV	Rd, Op2	数据传送	$Rd=Op2$
MVN	Rd, Op2	数据取反传送	$Rd=\sim Op2$
RSB	Rd, Rn, Op2	反向减法	$Rd=Op2-Rn$
RSC	Rd, Rn, Op2	带借位的反向减法	$Rd=Op2-Rn-!C$
SBC	Rd, Rn, Op2	带借位减法	$Rd=Rn-Op2-!C$
SUB	Rd, Rn, Op2	减法	$Rd=Rn-Op2$
逻辑运算			
AND	Rd, Rn, Op2	逻辑与	$Rd=Rn\&Op2$
BIC	Rd, Rn, Op2	位清除	$Rd=Rn\&\sim Op2$
EOR	Rd, Rn, Op2	逻辑异或	$Rd=Rn\wedge Op2$
ORR	Rd, Rn, Op2	逻辑或	$Rd=Rn Op2$ (OR NOT) $Rd=Rn \sim Op2$
比较指令			
CMP	Rn, Op2	比较	$Rn-Op2$
CMN	Rn, Op2	反值比较	$Rn+Op2$
TEQ	Rn, Op2	相等测试	$Rn\wedge Op2$
TST	Rn, Op2	位测试	$Rn\&Op2$

2) 第二操作数与移位

数据处理指令中的第二源操作数在指令编码中占用了 12 位。比较而言，第一源操作数和目的操作数的编码只有 4 个比特位，对应于 r0~r15 共 16 个寄存器。这样的编码安排使得第二源操作数具有更大的灵活性，可以实现不同的功能。总的来说，第二源操作数可以是一个带移位操作的寄存器或者作为立即数，指令编码中将通过第 25 位来区分第二源操作数是一个寄存器还是一个立即数。下面将分别讨论。

当第二源操作数是一个寄存器时，CPU 可以在将该寄存器的值送入 ALU 之前先对该寄存器的值进行移位操作。移位操作在 ARM 指令集中不作为单独的指令使用，它只能作为数据处理指令的一部分。这样设计的好处是可以在一个指令的执行周期中同时完成移位操作和数据处理操作。移位操作包括 6 种类型，ASL（算术左移）和 LSL（逻辑左移）是等价的，所以图 4-21 仅给出了 5 种移位操作。

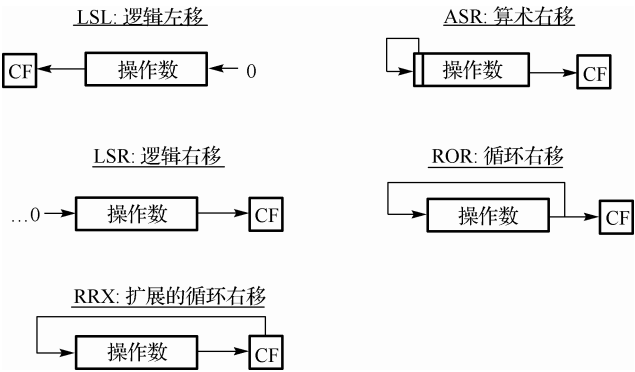


图 4-21 移位操作

我们可以使用移位操作来实现寄存器与简单常数的乘法或除法操作，如下面的例子：

```
RSB r2, r3, r3, LSL #4      ;r2 = r3 * 16-r3 = r3 * 15
RSB r2, r2, r2, LSL #3      ;r2 = r2*8 - r2 = r2 * 7
```

这个例子中，我们使用了 RSB 指令，所谓反向减法，也就是用第二源操作数减去第一源操作数（注意，SUB 指令是第一源操作数减去第二源操作数）。

当第二源操作数是立即数时，由于 ARM 是 RISC 架构，所有的 ARM 指令都是定常的 32 位，因此 ARM 无法在指令中表示任意 32 位立即数。第二源操作数占用 12 位编码，因此理论上这 12 位编码可以表示 0~4095 或者-2047~2047 的立即数。有研究发现 56%的常数在-15~+15 之间，98% 的常数在-511~+511 之间。请见扩展阅读[1]。虽然如此，在某些场合，程序还是需要使用 32 位立即数，比如某个数据区的基地址 0x38000000 等。基于这样的考虑，ARM 处理器采用了一个折中的解决办法：将 12 位编码分为两个部分，如图 4-22 所示，其中第 0 位~第 7 位作为立即数的种子 `immed_8`（可以表示范围 0~255 的立即数）；第 8 位~第 11 位作为移位因子 `rot`。最终指令执行时，送往 ALU 的第二源操作数实际上是 `immed_8` 循环右移 $2\times rot$ 位的结果。（如果你觉得循环右移比较难以理解，则可以将循环右移 $2\times rot$ 位转化为循环左移 $32-2\times rot$ 位，这两者是等价的。）

我们可以看下面这个例子。一条指令的编码为 0xE3A004FF，通过分析可以得知该指令是将一个立即数传送到 r0 寄存器的指令（Mov r0, #? ），编码中的 0x4FF 就是第二源操作数，其中立即数种子为 0xFF，移位因子为 0x4，因此该立即数是 0xFF000000（0xFF 循环右移 $2\times 4 = 8$ 位）。

显然采用立即数种子加移位的方式并不能生成所有的 32 位数，那么如果程序需要使用任意 32 位数该怎么办呢？因为 ARM 的指令中无法存放一个完整的 32 位数，因此只能将这个数存放在内存中的某个位置，程序执行的时候将其从内存中装载到相应的寄存器中，这就必须用到 LDR 指令。因此对于装载任意 32 位立即数，

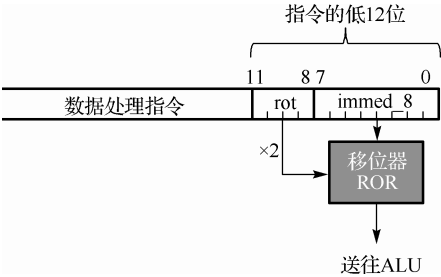


图 4-22 立即数的生成

ARM 的解决办法如下:

```
;以 PC 为基址, 向下偏移 imm12 (12 位偏移量), 装载常数 0x12345678
LDR r0, [PC, #imm12]
:
B Label1 ;程序必须跳过嵌在代码中的常量池
DCD 0x12345678
DCD 0x*****
:
Label1 SUB r0, r1, r2 ;其他指令
```

这种通过 LDR 指令装载任一 32 位常数的方法通常称为常量池 (Literal Pool)。常量池嵌在代码中的数据区, 虽然这有违于代码与数据相分离的原则, 但对于 RISC 架构的处理器而言, 这是没有办法的办法。注意, 在上面的例子中, LDR 指令使用当前 PC (其实是 LDR 指令的地址 + 8) 作为基址, 而不是采用一个绝对地址 (比如 0x40000000) 作为基址。这样做的好处是产生所谓的位置无关代码 (PIC, Position Independent Code), 也就是说不管这段代码被装载到内存的任何位置, LDR 指令与所访问的常量池之间的偏移量是固定的, 不需要在加载可执行代码后重新定位 (Relocation)。以 PC 作为基址进行偏移是 ARM 的一个特色, 我们将在跳转指令中再次看到基于 PC 的跳转指令。另一点需要说明的是, 在程序执行到常量池边界的时候, 必须跳转到正常的代码区, 如上面例子中的 “B Label1” 指令, 这条指令是必需的, 否则 CPU 将会错误地将常量池中的数据作为指令。

不管是立即数种子加移位的方法还是常量池的方法, 对于程序员而言都显得非常不方便, 因此 ARM 的汇编器通过一条伪指令提供了一个简便的方法来处理立即数的问题。

```
LDR r0, =0xFF ;汇编器将用 MOV r0, #0xFF 指令替换这条伪指令
LDR r0, =0x55555555 ;汇编器将用 LDR r0, [PC, #Imm12] 指令替换
```

上面例子中的 “LDR r0, =0xFF” 其实是一条伪指令, 注意立即数前面的符号是 “=” 而不是指令中的 “#”, ARM 汇编器在进行汇编时会自动判断 “=” 之后的立即数是否可以通过立即数种子加移位的方式生成, 如果可以则将该条伪指令汇编为 MOV 指令, 否则将其汇编为 LDR 指令。

常量池的方法最终是通过访存加载一个常量, 这需要使用 LDR 指令访存, 另外在常量池上边界还需要增加一条跳转指令, 这些都为程序的执行带来效率上的开销, 因此在最新的 ARM 架构中引入了 MOVW (Move Wide, 将 16 位数传送到目的寄存器的低 16 位并将高 16 位清零) 指令和 MOVT (Move Top, 将 16 位数传送到目的寄存器的高 16 位并保持低 16 位不变) 指令, 分别装载低 16 位和高 16 位。这样就可以通过两条指令完成任意 32 位立即数的加载。与常量池的方法相比, 这两条指令不需要访存, 且没有跳转指令。

```
MOVW R0, #:lower16:label
MOVT R0, #:upper16:label
```

3) 乘法指令与乘加指令

ARM 微处理器支持的乘法指令与乘加指令共有 6 条, 可分为运算结果为 32 位和运算结果为 64 位两类, 与前面的数据处理指令不同, 指令中的所有操作数 (包括源操作数和目的操作数) 必须为通用寄存器, 不能对操作数使用立即数或被移位的寄存器, 同时, 目的寄存器 Rd (不能为 PC) 和第一源操作数 Rn 必须是不同的寄存器。v4T 架构的乘法指令与乘加指令共有 6 条, 如表 4-11 所示。

表 4-11 乘法与乘加指令

操作码	操作数	描述	功能
MLA	Rd, Rn, Rm, Ra	32 位乘加	$Rd = Ra + (Rn * Rm)$
MUL	Rd, Rn, Rm	32 位乘法	$Rd = Rn * Rm$
SMLAL	RdLo, RdHi, Rn, Rm	64 位有符号数乘加	$RdHiLo += Rn * Rm$
SMULL	RdLo, RdHi, Rn, Rm	64 位有符号数乘法	$RdHiLo = Rn * Rm$
UMLAL	RdLo, RdHi, Rn, Rm	64 位无符号数乘加	$RdHiLo += Rn * Rm$
UMULL	RdLo, RdHi, Rn, Rm	64 位无符号数乘法	$RdHiLo = Rn * Rm$

MLA 指令和 MUL 指令不需要指定是有符号数（Signed）还是无符号数（Unsigned），这是因为这两条指令只将结果的低 32 位保存到 Rd 寄存器，因此不管源操作数的符号如何都不影响结果。对于 ARM7TDMI 和 ARM9TDMI 处理器而言，由于采用 8 位 Booth 算法，每个时钟周期只能计算 8 位，因此 MUL 和 MLA 指令的执行需要 2~5 个周期（如果乘数中有全 0 或全 1 的 8 位，则提前终止计算）。

4) 加载/存储指令

ARM 微处理器支持加载/存储指令用于在寄存器和存储器之间传送数据，加载指令用于将存储器中的数据传送到寄存器，存储指令则完成相反的操作。常用的加载/存储指令如下：

- LDR 字数据加载指令。
- LDRB 字节数据加载指令，目的寄存器高 24 位清零。
- LDRH 半字数据加载指令，目的寄存器高 16 位清零。
- LDRSB 有符号字节数据加载指令，目的寄存器高 24 位做符号位扩展。
- LDRSH 有符号半字数据加载指令，目的寄存器高 16 位做符号位扩展。
- STR 字数据存储指令。
- STRB 字节数据存储指令。
- STRH 半字数据存储指令。

早期的 ARM 架构中没有 LDRSB 和 LRDSH 这两条指令，程序员需要加载一个有符号 8 位数或 16 位数时需要自己做符号位扩展，请看下面的例子。

```
LDR r0, [r1, #0]           ;以 r1 的内容为基址，偏移量为 0，加载一个 32 位数到 r0
MOV r0, r0, LSL #24        ;r0 左移 24 位，将高 24 位无效数据移除
MOV r0, r0, ASR #24        ;r0 算术右移 24 位，做高 24 位的符号位扩展
```

因此装载一个有符号 8 位数需要执行 3 条指令，但有了 LDRSB/LDRSH 指令后只需要一条指令就可以完成这个工作（该指令会自动完成高 16 位或高 24 位的符号位扩展）。

```
LDRSB r0, [r1, #0]         ;以 r1+0 为地址，加载一个 8 位有符号数到 r0
```

因为存储的时候所有有符号数的高位都已经做好了符号位扩展，因此可以用 STDB/STRH 完成有符号数的存储，没有必要为有符号数设计专门的指令。同理，对于 32 位数而言，LDR/STR 在处理有符号数与无符号数时是相同的，也没有必要设计专门的 32 位有符号数的加载与存储指令。

图 4-23 给出了几条 LDR/STR 类指令的格式，可以看出这些指令的编码格式都比较接近，但所有涉及半字操作（LDRH、LDRSH、STRH）和有符号字节操作（LDRSB）的指令，其指令编码中的地址部分只有 8 位，而不像其他指令具有 12 位，这使得这些指令的寻址方式受到一定的限制。

LDR/STR 类指令的基本寻址方式可以归纳为寄存器基址加偏移量的寻址方式，但根据偏移量的生成方式，地址前变、后变还是带回写的前变等细节问题，LDR/STR 类指令的寻址方式可以细分为 9

种情况（对于半字操作和有符号字节操作，由于其指令编码中只有 8 位用来表示地址，因此这类指令的寻址方式只有 6 种），详见表 4-12。

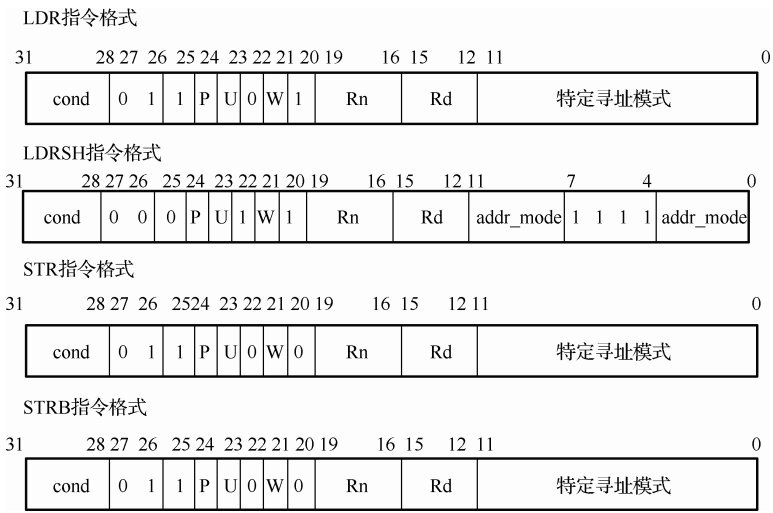


图 4-23 LDR 与 STR 类指令的格式

表 4-12 LDR/STR 类指令的寻址方式

字操作或无符号字节操作		半字操作或有符号字节操作	
寻址方式 1	语法 1	寻址方式 2	语法 2
立即数偏移前变址	[Rn, #+/-offset_12]	立即数偏移前变址	[Rn, #+/-offset_8]
寄存器偏移前变址	[Rn, +/- Rm]	寄存器偏移前变址	[Rn, +/- Rm]
比例寄存器偏移前变址	[Rn, +/- Rm, shift_imm]		
立即数偏移回写前变址	[Rn, #+/-offset_12]!	立即数偏移回写前变址	[Rn, #+/-offset_8]!
寄存器回写前变址	[Rn, +/- Rm]!	寄存器回写前变址	[Rn, +/- Rm]!
比例寄存器回写前变址	[Rn, +/- Rm, shift_imm]!		
立即数后变址	[Rn], #+/-offset_12	立即数后变址	[Rn], #+/-offset_8
寄存器后变址	[Rn], +/- Rm	寄存器后变址	[Rn], +/- Rm
比例寄存器后变址	[Rn], +/- Rm, shift_imm		

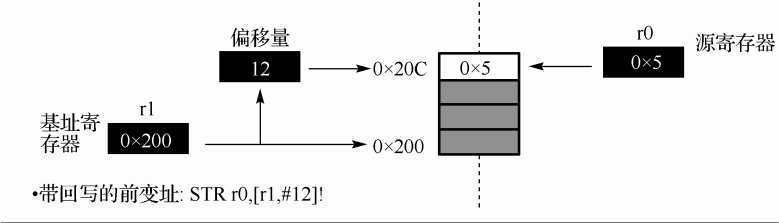
LDR/STR 类指令的偏移量可以有 3 种方式：12 位立即数（半字，有符号字节操作只能有 8 位立即数）；另一个寄存器；带移位操作的另一个寄存器（半字，有符号字节操作的指令没有这种偏移量）。根据地址生成的顺序，寻址方式又可以分为前变址、后变址与带回写的前变址。所谓前变址是指指令访问的内存地址 = 基址 + 偏移量，但指令访问结束后基址的内容不变；而后变址访问的内存地址 = 基址，访问结束后，基址 = 基址 + 偏移量；带回写的前变址所访问内存地址 = 基址 + 偏移量，访问结束后，基址 = 基址 + 偏移量。图 4-24 给出了前变址与后变址的区别（请注意指令助记符中偏移量在方括号里面还是后面）。

下面的例子给出了通过 LDR/STR 指令实现一个简单的内存复制的例子。注意，在这个例子中有一个假设，即所复制的内存大小是四字节对齐的。需要说明的是，如果所复制的内容不是 4 节对齐的，则需要采用 LDRB 和 STRB 一个字节一个字节地复制，显然这样的复制效率要低很多。

```
;r8 指向源数据区的首地址
;r9 指向源数据区的尾地址
```

```
;r10 指向目的数据区的首地址
loop    LDR r0, [r8], #4      ;装载 4 字节, 并更新 r8
        STR r0, [r10], #4    ;保存在 r10 指定的首地址, 并更新 r10
        CMP r8, r9           ;r8 到达结束地址了吗?
        BLT loop             ;没有到达结束则继续循环
```

• 前变址: STR r0, [r1, #12]



• 带回写的前变址: STR r0,[r1,#12]!

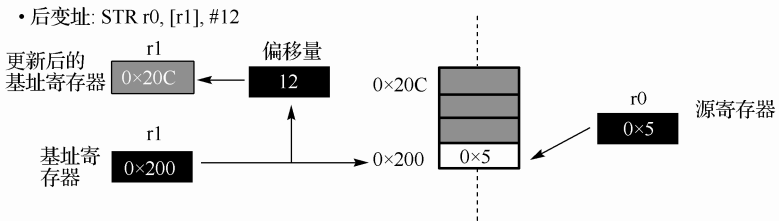


图 4-24 前变址与后变址的区别

5) 批量数据加载/存储指令

为了提高访存效率，ARM 微处理器支持批量数据加载（LDM）/存储（STM）指令，这些指令可以一次在一片连续的存储器单元和多个寄存器之间传送数据。批量加载指令用于将一片连续的存储器中的数据传送到多个寄存器，批量数据存储指令则完成相反的操作。LDM（或 STM）指令的格式为：

```
LDM（或 STM）{条件}{类型} 基址寄存器{!}，寄存器列表{^}
```

注意，所有的批量加载/存储指令必须指明加载类型，如图 4-25 所示，分为以下几种情况：

- IA Increment After，每次传送后地址加 4；
- IB Increment Before，每次传送前地址加 4；
- DA Decreament After，每次传送后地址减 4；
- DB Decreament Before，每次传送前地址减 4；
- FD Full Desceeds，满递减堆栈（等效于 DB）；
- ED Empty Descens，空递减堆栈（等效于 DA）；
- FA Full Ascends，满递增堆栈（等效于 IB）；
- EA Empty Ascends，空递增堆栈（等效于 IA）。

基址寄存器不允许为 R15，寄存器列表可以为 R0~R15 的任意组合。不管程序员在寄存器列表中给出的顺序如何，LDM/STM 指令都将按照低地址对应低寄存器、高地址对应高寄存器的方式进行存取，因此下面两条指令的执行结果是一样的。

```
STMIB r10, {r0, r1, r2, r3} ;r0->[r10+4], r1->[r10+8], r2->[r10+12], r3->[r10+16]
STMIB r10, {r3, r2, r1, r0} ;r0->[r10+4], r1->[r10+8], r2->[r10+12], r3->[r10+16]
```

{!} 为可选后缀，若选用该后缀，则当数据传送完毕之后，将最后的地址写入基址寄存器，否则基址寄存器的内容不改变。

{^}为可选后缀，当指令为 LDM 且寄存器列表中包含 R15，选用该后缀时表示：除了正常的数
据传送之外，还将 SPSR 复制到 CPSR。同时，该后缀还表示传入或传出的是用户模式下的寄存器，而
不是当前模式下的寄存器。

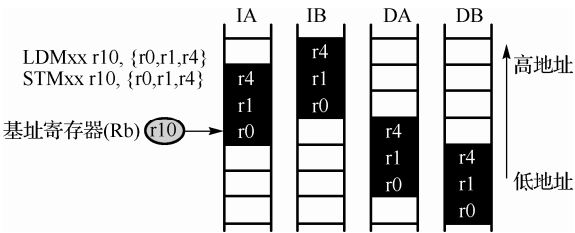


图 4-25 IA (EA)、IB (FA)、DA (ED)、DB (FD) 4 种不同类型的区别

程序员可以利用批量装载和存储指令完成大块内存的复制，下面的例子假设所复制的内容是按照
32 字节对齐的，在循环中每次都装载 32 字节到寄存器中，并在下一条指令中一次将这 32 字节写入目
的地址。显然，采用这种方式的复制效率要高于采用 LDR/STR 指令，比 LDRB/STRB 指令的效率就
要更高。虽然 LDM/STM 指令的执行时间不能在一个周期内完成（取决于装载/存储寄存器的个数），
但由于循环的次数少了，CPU 访存取指的次数也少了。另外，采用批量装载/存储指令还可充分利用总
线的 Busrt 特性，进一步提升了访存效率。

```
;r8 指向源数据区的首地址
;r9 指向源数据区的尾地址
;r10 指向目的数据区的首地址
Loop    LDMIA    r8!, {r0~r7}    ;装载 32 字节（8 个字）
        STMIA    r10!, {r0~r7}   ;存储 32 字节
        CMP     r8, r9           ;r8 到达结束地址了吗？
        BLT     loop            ;没有到达结束则继续循环
```

批量加载/存储指令的另一个用途就是作为堆栈操作指令。程序员可以利用这类指令一次性完成若
干寄存器的压栈操作和退栈操作，如图 4-26 所示。图中左边的指令将 r4~r7 和 r14 (lr) 共 5 个寄存
器压入堆栈，其中 sp 通常为 r13。注意，该指令执行完后需要更新 sp 寄存器的值 (sp!)。右边的指
令是退栈指令，将当前堆栈的内容弹出到 r4~r7 和 r15 (PC) 共 5 个寄存器中，并更新 sp。

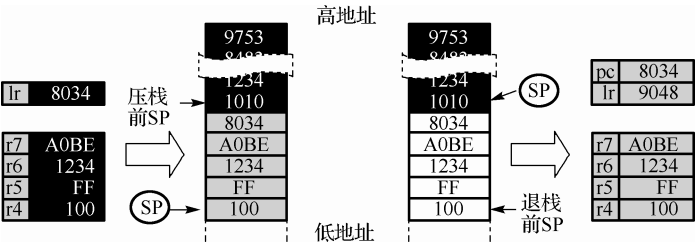


图 4-26 堆栈操作（满递减栈）指令

6) 数据交换指令

数据交换指令能在存储器和寄存器之间交换数据。数据交换指令有如下两条：

- SWP 字数据交换指令。
- SWPB 字节数据交换指令。

SWP 指令的格式是：

SWP{<条件>}{B} Rd, Rm, [Rn]

SWP 指令的过程首先以 Rn 为基址，将其所指向内存中的值（32 位或 8 位，取决于指令是 SWP 还是 SWPB）加载到一个临时寄存器，然后将 Rm 的值写入 Rn 指向的地址中，最后将临时寄存器的值传送到 Rd 寄存器，如图 4-27 所示。如果 Rm 和 Rd 寄存器是同一个寄存器，则该指令完成内存中的某个值与目的寄存器值的交换。以上操作在一条指令中完成，这个数据交换的过程是一个原子过程（即不可以被中断），使用这条指令可以实现操作系统中的信号量，且不需要关中断。SWP 指令是 ARM 中唯一一条非 LDR/STR 类指令而访存的指令。SWP 指令不能由编译器自动生成，因此程序员如果要使用该指令，只能通过手工汇编的方式。

7) 跳转指令

跳转指令用于实现程序流程的跳转，在 ARM 程序中有两种方法可以实现程序流程的跳转：

- 使用专门的跳转指令。
- 直接向程序计数器 PC 写入跳转地址值。

ARM 指令集中的跳转指令可以完成从当前指令向前或向后的 32MB 的地址空间的跳转，包括以下 4 条指令：

- B 跳转指令。
- BL 保存返回地址的跳转指令，用于子函数的调用（相当于 X86 的 Call 指令）。
- BLX 保存返回地址和状态切换的跳转指令。
- BX 带状态切换的跳转指令，用于在 ARM 和 Thumb 状态间切换。

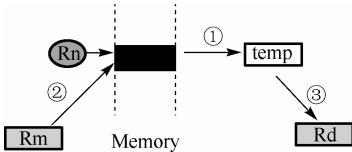


图 4-27 指令的操作顺序

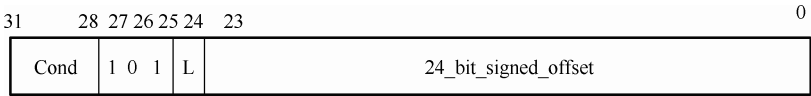


图 4-28 B (L) 指令的编码格式

从图 4-28 中可以看出，对于 B 和 BL 指令而言，其跳转偏移量是一个有符号 23 位数（最高位是符号位），由于跳转只能发生在 4 字节对齐的地址上，因此 23 位偏移量实际上可以在最低两位扩展为以 0b00 结尾的 25 位偏移量，这样加上符号位可以表示为-32MB~+32MB 的地址范围。与 X86 系统中的跳转基本上以绝对地址作为目的地址不同，ARM 的跳转指令全部采用相对地址（基址+偏移量），跳转指令中并没有指定基址寄存器，因为所有的跳转都以当前 PC（其实是当前跳转指令地址+8）作为基址。以 PC 作为基址的好处是可以使生成的代码做到位置无关（PIC），程序加载到内存时重定位（Relocation）的压力会很轻。程序员并不需要自己计算目的地址与当前跳转指令之间的偏移量，只需要在汇编代码中使用标号（Label），汇编器会自动计算该 Label 与跳转指令之间的距离，并将该距离作为偏移量填入跳转指令。

对于需要大于 32MB 的长跳转，可以通过下面两种方法实现。

(1) MOV PC,#imme_value

把目标地址直接赋给 PC 寄存器。

但是这条指令受格式限制并不能处理任意立即数，只有当这个立即数能够表示为一个 8 位数值通过循环右移偶数位而得到时才是合法的。例如，MOV PC,#0x30000000 是合法的，因为 0x30000000 可以通过 0x03 循环右移 4 位而得到。而 MOV PC,#0x30003000 就是非法指令。

(2) LDR PC,[PC+offset]

把目标地址先存储在某一个合适的地址空间,然后把这个存储器单元上的 32 位数据传送给 PC 来实现跳转。这种方法对目标地址值没有要求,可以是任意有效地址。但是存储目标地址的存储器单元必须在当前指令的 4KB (11 位有符号偏移量) 空间范围内。

8) 程序状态寄存器访问指令

ARM 微处理器支持程序状态寄存器访问指令,用于在程序状态寄存器和通用寄存器之间传送数据,程序状态寄存器访问指令包括以下两条:

- MRS 程序状态寄存器到通用寄存器的数据传送指令。
- MSR 通用寄存器到程序状态寄存器的数据传送指令。

9) 协处理器指令

ARM 微处理器可支持多达 16 个协处理器,用于各种协处理操作,在程序执行的过程中,每个协处理器只执行针对自身的协处理指令,忽略 ARM 处理器和其他协处理器的指令。

ARM 的协处理器指令主要用于 ARM 处理器初始化 ARM 协处理器的数据处理操作,以及在 ARM 处理器的寄存器和协处理器的寄存器之间传送数据,以及在 ARM 协处理器的寄存器和存储器之间传送数据。ARM 协处理器指令包括以下 5 条:

- CDP 协处理器数据操作指令。
- LDC 协处理器数据加载指令。
- STC 协处理器数据存储指令。
- MCR ARM 处理器寄存器到协处理器寄存器的数据传送指令。
- MRC 协处理器寄存器到 ARM 处理器寄存器的数据传送指令。

10) 异常产生指令

SWI 指令用于产生软件中断,以使用户程序能调用操作系统的系统例程。操作系统在 SWI 的异常处理程序中提供相应的系统服务,指令中 24 位的立即数指定用户程序调用系统例程的类型,相关参数通过调用寄存器传递,当指令中 24 位的立即数被忽略时,用户程序调用系统例程的类型由通用寄存器 R0 的内容决定,同时,参数通过其他通用寄存器传递。

指令示例:

SWI 0x02 ; 该指令调用操作系统编号为 02 的系统例程

我们将在 4.2.5 节详细介绍 ARM 处理器的中断处理。

4.2.4 ARM7TDMI 汇编语言

1. 汇编语言的程序结构

在 ARM (Thumb) 汇编语言程序中,以程序段为单位组织代码。段是相对独立的指令或数据序列,具有特定的名称。段可以分为代码段和数据段,代码段的内容为执行代码,数据段存放代码运行时需要用到的数据。一个汇编程序至少应该有一个代码段,当程序较长时,可以分割为多个代码段和数据段,多个段在程序编译链接时最终形成一个可执行的映像文件。

可执行映像文件通常由以下几部分构成:

- 一个或多个代码段,代码段的属性为只读,简称 RO 段 (Read Only)。
- 零个或多个包含初始化数据的数据段,数据段的属性为可读写,简称 RW 段 (Read and Write)。
- 零个或多个不包含初始化数据的数据段,数据段的属性为可读写,简称 ZI 段 (Zero Initialized)。

链接器根据系统默认或用户设定的规则,将各个段安排在存储器中的相应位置。因此,源程序中

段之间的相对位置与可执行的映像文件中段的相对位置一般不会相同。

以下是一个汇编语言源程序的基本结构：

```
AREA Init, CODE, READONLY
ENTRY
Start
LDR    R0, =0x3FF5000
LDR    R1, 0xFF
STR    R1, [R0]
LDR    R0, =0x3FF5008
LDR    R1, 0x01
STR    R1, [R0]
:
END
```

在汇编语言程序中，用 AREA 伪指令定义一个段，并说明所定义段的相关属性，本例定义一个名为 Init 的代码段，属性为只读。ENTRY 伪指令标识程序的入口点，接下来为指令序列，程序的末尾为 END 伪指令，该伪指令告诉编译器源文件的结束，每一个汇编程序段都必须有一条 END 伪指令，指示代码段的结束。

2. 汇编语言的子程序调用

在 ARM 汇编语言程序中，子程序的调用一般是通过 BL 指令来实现的。在程序中，使用指令：

```
BL 子程序名
```

即可完成子程序的调用。

该指令在执行时完成如下操作：将子程序的返回地址存放在连接寄存器 LR 中，同时将程序计数器 PC 指向子程序的入口点，当子程序执行完毕需要返回调用处时，只需要将存放在 LR 中的返回地址重新复制给程序计数器 PC 即可。在调用子程序的同时，也可以完成参数的传递和从子程序返回运算的结果，通常传参可以使用寄存器 R0~R3 完成，而返回值则应该在返回前存入 R0 寄存器。

以下是使用 BL 指令调用子程序的汇编语言源程序的基本结构：

```
AREA    Init, CODE, READONLY
ENTRY
Start
LDR     R0, =0x3FF5000
LDR     R1, 0xFF
STR     R1, [R0]
LDR     R0, =0x3FF5008
LDR     R1, 0x01
STR     R1, [R0]
BL      PRINT_TEXT
:
PRINT_TEXT
:
MOV     PC, LR
:
END
```


3. 汇编语言程序示例

以下是系统初始化部分的代码，该段代码建立了异常向量表，提供了部分复位异常处理程序。读者可以根据这段程序来理解一个完整汇编语言程序的基本结构、各种指令的运用、异常处理程序的实现等。

```
include hardware.inc
extern main

AREA BOOT, CODE, READONLY
ENTRY                                ;程序的入口点，一个程序至少要有个 ENTRY
;异常向量表
bal      RST DO
bal      EXTENT INSTRU
bal      SWI DO
bal      ABORT PREFETCH DO
bal      ABORT DATA DO
mov      R1,R1                        ;保留
bal      Irq_Do
Fiq_Handler                          ;此处可以放置 FIQ 中断处理程序，加快 FIQ 中断处理速度

RST_DO
EXPORT   RST_DO

;初始化所有处理器模式下的堆栈指针
ldr      sp, =SP_SVC                  ;复位异常进入管理模式 (svc)，初始化管理模式的堆栈指针
mov      r0, #0xD2
msr      cpsr_cf, r0                  ;切换到 IRQ 模式，初始化 IRQ 模式的堆栈指针
ldr      sp, =SP_IRQ

mov      r0, #0xD1
msr      cpsr_cf, r0                  ;切换到 FIQ 模式，初始化 FIQ 模式的堆栈指针
ldr      sp, =SP_FIQ

mov      r0, #0xD7
msr      cpsr_cf, r0                  ;切换到中止模式 (abt)，初始化中止模式的堆栈指针
ldr      sp, =SP_ABT

mov      r0, #0xDB
msr      cpsr_cf, r0                  ;切换到未定义模式 (und)，初始化未定义模式的堆栈指针
ldr      sp, =SP_UND

mov      r0, #0xDF
msr      cpsr_cxsf, r0                ;切换到系统模式 (sys)，初始化系统模式的堆栈指针
ldr      sp, =SP_SYS

mov      r0, #0xD3
msr      cpsr_c, r0                  ;切换到管理模式 (svc)，代码通常运行在管理模式

mrs      r0, cpsr
bic      r0, r0, #0x80                ;使能 IRQ 中断
msr      cpsr_c, r0
```

```
IMPORT    __main
b        __main                ;跳转到 C 语言的主程序运行，不再回来

;*****
AREA DATA_SECTION, DATA, READWRITE
;这是一个可读可写的数据段，段的名字是 DATA_SECTION
;DCD 伪指令用于分配一片连续的字存储单元并用指定的数据初始化。
LR_USR      DCD      0X0
LR_SYS      DCD      0X0
LR_SVC      DCD      0X0
LR_IRQ      DCD      0X0
LR_FIQ      DCD      0X0
LR_UND      DCD      0X0
LR_ABT      DCD      0X0

END                ;告诉编译器已经到了源程序的结尾，源程序必须以 END 结尾
```

4.2.5 ARM7TDMI 异常处理

所谓异常（Exception），广义来说就是处理器的中断。处理器的异常（或中断）可以分为 3 类：硬件中断、软件中断和异常。硬件中断（Hardware Interrupt）一般是由外部（相对 CPU 内核而言）的硬件引起的事件，比如串口来数据、键盘击键等；软件中断（Software Interrupt）是通过在程序中执行的中断指令引起的中断，又称为软陷指令，如 80x86 的 int 指令、68000 的 trap 指令、ARM 中的 SWI 指令。软中断指令一般用于操作系统的系统调用入口。异常是由于 CPU 内部在运行过程中引起的事件，比如指令预取错、数据中止、未定义指令等，异常事件一般由操作系统接管。本书中对“异常”和“中断”不做严格区分，都是指请求处理器打断正常的程序执行流程，进入特定服务程序的一种机制。

ARM 体系结构有 7 种异常：复位异常、未定义指令异常、软件中断、指令预取中止异常、数据中止异常、FIQ 中断、IRQ 中断。本书中把复位异常、未定义指令异常、指令预取中止异常、数据中止异常称为异常，把软件中断、FIQ 中断、IRQ 中断称为中断。

在处理异常之前，当前处理器的状态必须保留，这样当异常处理完成之后，当前程序可以继续执行。处理器允许多个异常同时发生，它们将会按固定的优先级进行处理。

1. ARM 体系结构所支持的异常类型

ARM 体系结构所支持的异常类型及具体含义如表 4-13 所示。

表 4-13 ARM 体系结构所支持的异常

异常类型	具体含义
复位（Reset）	当处理器的复位电平有效时，产生复位异常，程序跳转到复位异常处理程序处执行
未定义指令 （Undefined Instruction）	当 ARM 处理器或协处理器遇到不能处理的指令时，产生未定义指令异常。可使用该异常机制进行软件仿真
软件中断 SWI （Software Interrupt）	该异常由执行 SWI 指令产生，用户模式下执行该指令，可以使 CPU 工作模式切换到特权模式。操作系统可使用该异常机制实现系统调用
指令预取中止 （Prefech Abort）	若处理器预取指令的地址不存在，或该地址不允许当前程序访问，会触发指令预取异常。只有当产生该异常的指令进入执行阶段时，CPU 才会响应指令预取中止异常
数据中止 （Data Abort）	若处理器数据访问指令的地址不存在，或该地址不允许当前指令访问时，产生数据中止异常

续表

异常类型	具体含义
IRQ（外部中断请求）	当 CPU 的外部中断请求引脚有效，且 CPSR 中的 I 位为 0 时，产生 IRQ 异常。系统的外设可通过该异常请求外部硬件中断服务
FIQ（快速中断请求）	当 CPU 的快速中断请求引脚有效，且 CPSR 中的 F 位为 0 时，产生 FIQ 异常

2. 异常向量表及优先级

异常向量表中指定了各异常及其处理程序的对应关系，它通常存放在存储器地址的低端（0x00000000，ARM720T 以后的处理器也可以将异常向量表放在 0xFFFF0000）。在 ARM 体系中，异常向量表的大小为 32 字节（见图 4-29）。其中，每个异常向量占据 4 个字节大小，保留了 4 个字节空间。

每个异常对应了异常向量表中的 4 个字节的空间，其中存放了一个跳转指令或者一个向 PC 寄存器中赋值的数据访问指令。通过这两种指令，程序将跳转到相应的异常处理程序处执行。需要注意的是，很多处理器（比如 x86）通常在异常向量表中直接存放的是中断处理程序的入口地址，而 ARM 却是在向量表中存放指令（这其实也是符合 ARM 的 Load/Store 体系结构原则的，因为如果中断向量表中存放的是地址，则意味着 CPU 将由硬件读取存储器中的值，并将读取的值传递给 PC，显然这有背 Load/Store 原则）。

具体来说，存储在中断向量表中的指令可以有 4 种不同的方式（如图 4-30）所示，第一是最直接的 B 指令，但 B 指令的最大问题是存在 32MB 的范围限制（当然也可以在 32MB 范围的某个地址再存放另外一条 B 指令，这样就可以通过“二级跳”的方式突破 32MB 边界）；第二，对于能够用立即数种子移位生成的目的地址可以直接 Mov 给 PC 以实现跳转，如图中的 MOV PC, #0x30000000；第三，对于任意 32 位地址，可以通过常量池将该任意地址存放其中，并通过一条 LDR 指令将其装载到 PC 中，如图中的 LDR PC, [PC, #FF0]，将存放在 0XFFC 地址中的 0x30008000 这个地址装载到 PC 中；第四，对于 FIQ，由于该向量的位置位于最后，因此可以在向量表及以后的一段连续地址内直接存放 FIQ 的处理程序，这样可以减少一次跳转，进一步减少了由于跳转而引起的性能损失。

当多个异常同时发生时，CPU 根据固定的优先级决定异常的处理次序。当然，有些异常是不可能同时发生的，如指令预取中止异常和数据终止异常是不可能同时发生的。各异常的向量地址以及异常的处理优先级见表 4-14。



图 4-29 ARM CPU 的异常向量表

地址	异常	进入模式	优先级（6 最低）
0x0000,0000	复位	管理模式	1
0x0000,0004	未定义指令	未定义模式	6
0x0000,0008	软件中断	管理模式	6
0x0000,000C	中止（预取指令）	中止模式	5
0x0000,0010	中止（数据）	中止模式	2
0x0000,0014	保留	保留	未使用
0x0000,0018	IRQ	IRQ	4
0x0000,001C	FIQ	FIQ	3

表 4-14 异常向量表

3. ARM 处理器对异常的响应过程

当一个异常出现以后，ARM 微处理器会执行以下几步操作：

- ① 保存处理器当前状态，中断屏蔽位以及各条件标志位。这是通过将当前程序状态寄存器 CPSR 的内容保存到将要执行的异常对应的 SPSR 寄存器中实现的。各异常有自己的物理 SPSR。
- ② 设置当前程序状态寄存器 CPSR 中的相应位。包括①进入 ARM 状态；②设置 CPSR 中的 mode 位，使处理器进入相应的执行模式；③设置 CPSR 中相应的 I 位，禁止 IRQ 中断，当进入 FIQ 模式时，禁止 FIQ 中断。
- ③ 将返回地址保存到寄存器 lr_mode。
- ④ 将程序计数器值（PC）设置成该异常的异常向量地址，从而跳转到相应的异常处理程序处执行。到这里控制权已经交给了异常处理软件。

需要说明的是以上这些操作是完全由硬件自动完成的，在这个响应过程中，程序员不可干预硬件的行为。

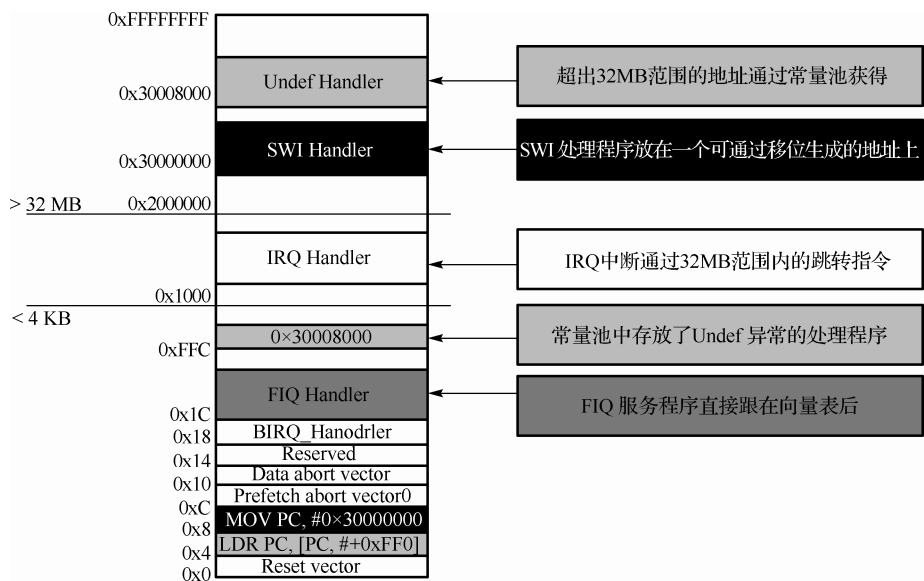


图 4-30 异常向量表中的指令

4. 从异常返回

当一个异常处理返回时，一共有 3 件事情需要处理：通用寄存器的恢复（如果异常服务程序需要使用其他寄存器，安全的方法是在使用前首先将其压栈保存，而在退出异常服务前将其出栈恢复）、状态寄存器的恢复以及 PC 指针的恢复。可以认为应用程序总是从复位异常处理程序开始执行的，因此复位异常处理程序不需要返回。

从异常处理程序中返回包括下面两个基本操作：

- (1) 恢复被中断的程序处理器的状态，即将 SPSR_mode 寄存器的内容复制到当前程序状态寄存器 CPSR 中。
- (2) 返回到发生异常的下一条指令处执行（对于预取终止异常和数据访问终止异常在返回的时候会往往会重新执行引起异常的这条指令），即将 lr_mode 寄存器的内容复制到程序计数器 PC 中。

与大多数其他处理器不同，ARM 处理器没有专门的中断返回指令（x86 处理器的 RETI 指令），

而且根据所进入的异常不同,其返回时的操作也不尽相同。总的来说,ARM 处理器的中断返回可以分为 3 种情况:

第一,SWI 异常与未定义指令异常。通常采用 `MOVS PC, Lr` 指令来作为中断的返回指令。在特权模式下,数据处理指令加“S”并且目的寄存器为 PC 时,除了更新 PC 的值外,还将恢复 SPSR 到 CPSR 中。Lr 寄存器中保存的是 SWI 指令或未定义指令之后的下一条指令地址(在流水线 A 周期, CPU 将调整 Lr 的值,使其指向该地址,参见 4.2.2 节),而这恰恰是中断返回后需要执行的指令地址,因此对于第一种情况,只需要简单地将 Lr 寄存器恢复到 PC,并同时恢复 SPSR 即可完成中断的返回。

第二,IRQ、FIQ 与指令预取终止异常。通常采用 `SUBS PC, LR, #0x4` 指令作为中断返回指令。与第一种情况相同,以 PC 为目的寄存器的 SUBS 指令在特权模式将恢复 SPSR 到 CPSR。对于 IRQ 和 FIQ 中断而言,只有当前指令执行结束前, CPU 才会响应来自 IRQ 和 FIQ 引脚的中断请求,而这时 PC 的值已经更新到当前指令地址偏移 `0xC` 的地址(通常情况下 PC 的值等于当前被执行指令的地址加 `0x8`,对于 IRQ 和 FIQ 由于当前指令已执行完毕,因此 PC 已被更新),这时保存在 Lr 中的地址就是当前指令地址加 `0x8`(因为 Ajust 周期会将 PC 的值减 4 后保存到 Lr),这个地址指向了当前指令后的第二条指令,如果简单地将 Lr 恢复到 PC 将漏掉一条指令。为了避免这种情况的发生,我们必须将 Lr 的值减去 4 之后再恢复到 PC。对于指令预取终止异常,虽然采用同样的指令进行异常返回,但与 IRQ/FIQ 的机制并不相同。指令预取异常在流水线的预取周期(F 周期)就会被 CPU 发现,但 CPU 仅标注这条指令为预取异常,并不在该周期响应这个异常,而是直到该指令进入了执行周期(E 周期)才响应该异常,此时 PC 的值等于当前指令地址加 `0x8`,因此 Lr 的值经过调整周期(A 周期)后为当前指令加 `0x4`。与其他异常返回不同的是,对于预取终止异常,在返回的时候往往需要重新执行引起该异常的指令(预取异常可能是因为虚存机制还未将该指令所在内存映射而造成的,因此在操作系统接管该异常并将该页内存映射后,需要重新执行该指令)。为了能够重新执行该指令,需要将 Lr 的值再减 4 后恢复到 PC。

第三,对于数据访问终止异常,通常采用 `SUBS PC, Lr, #0x8` 指令作为异常返回指令。当 LDR/STR 类指令访问存储器时(M 周期)才有可能发现数据访问终止,而这时的 PC 值已经更新到当前访存指令加 `0xC`,Lr 中的值经过调整后为当前访存指令地址加 `0x8`。与指令预取终止异常类似,由于数据访问终止异常往往是因为虚存机制尚未映射的原因,在异常处理中完成重新映射后,需要重新执行该访存指令以完成数据的访问。将 Lr 的值减 8 后恢复到 PC, CPU 所执行的指令将正好是该数据访问指令。

另外,如果程序员在进入异常服务程序时,则通过堆栈保存返回地址(大多数情况下,程序员都是这么做的),也可以通过出栈操作来进行异常返回,比如采用指令 `LDMFD sp!, {pc}^`。注意:①程序员需要保证堆栈中的返回地址已经是经过调整的正确地址(是 Lr,还是 Lr-4 或者 Lr-8)。这也就意味着返回地址在入栈前就必须首先调整为正确的地址;②指令中的“^”是不可少的,这个符号告诉处理器同时恢复 SPSR 到 CPSR;③因此在调用该指令前,程序员有必要将保存在堆栈中的 SPSR 值首先恢复到 SPSR 中。

5. SoC 中的中断处理

ARM 内核只有两个外部中断输入信号 nFIQ 和 nIRQ,但对于一个系统来说,中断源可能多达几十个。在 SoC 设计中,一般都会有一个中断控制器来处理多个中断信号(关于中断控制器读者也可以参阅 2.4 节),如图 4-31 所示。

对于采用中断控制器的系统,其中断处理的一般过程为:

(1) 外部中断源(比如串口 UART)发去中断请求,该请求信号被送到中断控制器进行仲裁与相关处理后(比如该信号是否被中断控制器所屏蔽),最终由中断控制器发出给 CPU 的中断请求信号 nIRQ。

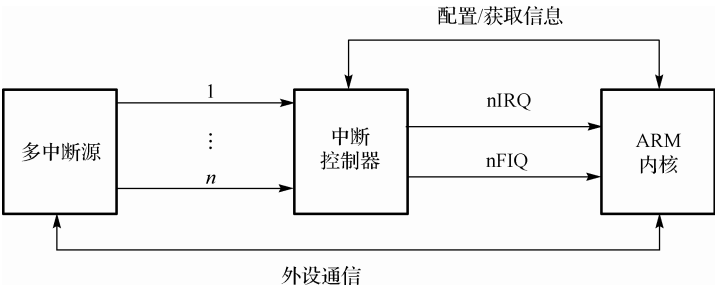


图 4-31 SoC 中的中断处理

(2) CPU 响应 IRQ 信号，并将控制权交给系统中断处理程序。系统中断处理程序在完成相应的现场保护工作后，读取中断控制器相应的状态寄存器（通常是一条 LDR 指令，因为对于以 ARM 处理器为内核的 SoC 而言，所有外设的寄存器都是与内存统一编址的），从该状态寄存器的值，中断处理程序可以获知该中断请求是由 UART 发出的，在清除中断控制器的中断请求后（有些 SoC 设计要求程序显式地清除中断请求，一般通过将中断控制器的某个控制位清零。有些 SoC 的设计则在程序读取中断状态后自动清除中断请求），调用 UART 的中断处理程序，如图 4-32 所示。注意，清中断的步骤是必需的，否则在完成本次中断处理后，中断控制器送往 CPU 的 nIRQ 信号将一直有效，造成一出本次中断后紧接着又再次进入中断的错误。

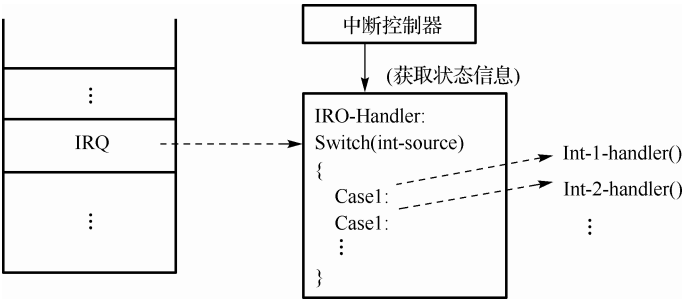


图 4-32 软件控制中断分支

(3) UART 的中断处理程序在获得控制权后，首先要读 UART 模块的相应寄存器（还是通过 LDR 指令），以确定产生本次中断的具体原因（UART 中断可能是接收 FIFO 满，或者接收缓冲区溢出，或者接收错等），并调用相应的处理程序，完成对本次中断的服务。注意，与清除中断控制器中断请求相同，UART 中断程序在调用具体的服务程序前，需要清除 UART 模块的中断请求，以避免重复进入中断。

(4) UART 中断处理程序将控制权交回系统中断处理程序，该程序在完成现场恢复后从中断返回。以下的代码给出了以 ARM 处理器为内核的 SoC 在处理中断时所需要的流程。

```
//模块中断处理程序的入口地址(函数指针)
void(*IntHandler[32])(void)={
    /*interrupt number and description, handler */
    /* 00 INT_NULL, */ ENT_INT_EMPTY ,
    /* 01 INT_EXT0, (PE0) */ ENT_INT_RING1 ,
    /* 02 INT_EXT1, (PE1) */ NULL ,
    /* 03 INT_EXT2, (PE2) */ NULL ,
    /* 04 INT_EXT3, (PE3) */ ENT_INT_RING2 ,
    /* 05 INT_EXT4, (PE4) */ NULL ,
}
```

```

/* 06 INT_EXT5, (PE5) */          NULL          ,
/* 07 INT_EXT6, (PE6) */          ENT_INT_RING3   ,
/* 08 INT_EXT7, (PE7) */          NULL          ,
/* 09 INT_EXT8, (PE8) */          NULL          ,
/* 10 INT_EXT9, (PE9) */          ENT_INT_BUTTON  ,
/* 11 INT_EXT10, (PE10) */        NULL          ,
/* 12 INT_EXT11, (PE11) */        NULL          ,
/* 13 INT_EXT12, (PH0) */         NULL          ,
/* 14 INT_EXT13, (PH1) */         ENT_INT_SSRT    ,
/* 15 INT_EXT14, (PH2) */         NULL          ,
/* 16 INT_NONE, */                NULL          ,
/* 17 INT_EXT15, (PH3) */         NULL          ,
/* 18 INT_EXT16, (PH4) */         NULL          ,
/* 19 INT_EXT17, (PH5) */         NULL          ,
/* 20 INT_LCDC, */                NULL          ,
/* 21 INT_AC97, */                NULL          ,
/* 22 INT_PWM, */                 NULL          ,
/* 23 INT_UART1, */               NULL          ,
/* 24 INT_UART0, */               NULL          ,
/* 25 INT_MMC, */                 NULL          ,
/* 26 INT_SPI, */                 NULL          ,
/* 27 INT_USB, */                 ENT_INT_USB     ,
/* 28 INT_GPT, */                 ENT_INT_GPT     ,
/* 29 INT_EMI, */                 ENT_INT_EMI     ,
/* 30 INT_DMA, */                 ENT_INT_DMA     ,
/* 31 INT_RTC, */                 ENT_INT_RTC     ,
};

//IRQ 系统中断处理程序将调用本函数，本函数通过检查中断控制器，找到相应的中断源，
//并调用相应的模块中断服务程序。
void int_vector_handler(void)
{ int i;

  unsigned long IFSTAT=*(RP) (INTC_IFSTAT); //读中断控制器的状态寄存器
  if(IFSTAT<1) IFSTAT=*(RP) (INTC_ISTAT) & (~*(RP) (INTC_IMSK))
                                     &*(RP) (INTC_IEN);

  if(IFSTAT>1)
  { i = -1;
    while(IFSTAT) //查找中断源
    { IFSTAT>>=1;
      i++;
    }
  }else i=0;

  if(IntHandler[i])
  (*IntHandler[i])(); //调用相应的模块中断处理程序
  else
  { ent_int();
    printf("No interrupt entry for INT NO.%d\n",i);
  }
}

```

```

    ret_int();
}
}

```

IntHandler[] 是用户自己开辟的一块存储器空间，里面按次序存放异常处理函数的地址。int_vector_handler() 函数从中断控制器读取中断源信息，然后再从 IntHandler[] 中的对应地址单元得到异常处理函数的入口地址，完成一次异常响应的跳转。这种方法的好处是用户程序在运行过程中能够很方便地动态改变异常服务内容。

6. 中断的嵌套

几乎所有的 CPU 在响应中断的时候都会将程序状态字中的中断屏蔽位置位，以防止在响应中断的过程中被其他中断干扰，因此缺省状态下处理器是不允许中断嵌套的。当然，在完成响应中断的前期工作后，程序员可以显式地打开中断使能，使得在中断处理中间接收其他的中断，也就是中断嵌套。几乎所有的实时操作系统都要求能够接收中断嵌套（虽然这会使操作系统的中断处理和任务调度变得更加复杂），因为如果在中断服务期间不接受其他高优先级中断，有可能会丢失或延误重要的硬件事件。

对于所有的 ARM 处理器而言，打开中断嵌套将有可能导致 Lr 寄存器的内容被意外覆盖。假设有这样一种情况，当前 CPU 已经在一个 IRQ 异常处理程序中，异常处理程序已经将当前的 Lr 寄存器值（也就是该中断的返回地址）保存到了堆栈中，并且使能了中断，之后中断处理程序通过 BL 指令调用了 C 函数，而恰好在 C 程序将 Lr（这时该 Lr 寄存器中保存了 C 函数的返回地址）压栈之前又产生了一个新的 IRQ 异常（注意这时 CPU 已经使能了中断），CPU 响应该异常，并且将这个新异常的返回地址保存到了 IRQ 模式下的 Lr 寄存器中，这就使得保存在该寄存器中的 C 函数返回地址被覆盖了。为了解决这个问题，ARM v4 体系架构以后的 CPU 引入了 System 模式，程序员只需要在使能中断前将 CPU 当前的工作模式切换到 System 模式下，就可以安全地执行 BL 指令了。因为在进入 System 模式并调用 BL 指令后（注意，这时 BL 的返回地址保存在 System 模式下的 Lr 寄存器），即使再次发生 IRQ 中断，该中断的返回地址也是保存在 IRQ 模式下的 Lr 寄存器中，不会覆盖 System 模式下的 Lr 寄存器。使用 System 模式的好处是，任何中断或异常都不会导致 CPU 进入该模式。当然，如果在中断处理程序中不使用 BL 指令，则可以不切换 CPU 工作模式。

另外一点需要注意的是，在从堆栈恢复 SPSR 的值到一个临时寄存器之前，必须先关闭中断嵌套，否则从堆栈弹出的 SPSR 的值保存在一个临时寄存器中，而被写回 SPSR 之前，有可能因为另一个中断的发生而被意外覆盖。下面的代码给出了可嵌套的中断处理程序的一个例子。

```

IRQHandler
SUB    lr, lr, #4                ;更新 lr 的值
STMFD  sp!, {lr}                ;Lr 入栈
MRS    r14, SPSR
STMFD  sp!, {r12, r14}          ;r12,SPSR 入栈

MOV     r12, #IntBase
LDR     r12, [r12, #IntSource]   ;读中断源并清中断

;将当前 CPU 模式切换到 System 并打开中断 IRQ 使能
MRS     r14, CPSR
BIC     r14, r14, #0x9F

```



```

    ORR    r14, r14, #0x1F
    MSR    CPSR_c, r14

    STMFD  sp!, {r0-r3, lr}           ;r0-r3 入栈, System 模式下的 Lr 入栈
    MOV    r0, r12                    ;r12 中存放着中断源号, 作为参数传给 C 程序
    BL C_irq_handler                  ;调用 C 函数
    LDMFD  sp!, {r0-r3, lr}

;切换回 IRQ 模式, 并关闭中断
    MRS    r12, CPSR
    BIC    r12, r12, 0x1F
    ORR    r12, r12, 0x92
    MSR    CPSR_c, r12
    LDMFD  sp!, {r12, r14}           ;恢复 r12, 并将保存的 SPSR 值恢复到 r14
    MSR    SPSR_csfxf, r14          ;将 r14 的值恢复到 irq 模式下的 SPSR
    LDMFD  sp!, {PC}^               ;返回地址出栈到 PC, SPSR 恢复到 CPSR
    
```

4.2.6 ARM 汇编程序与 C 程序

1. ATPCS 概述

为了使单独编译的 C 语言和汇编语言之间能够相互调用, 必须为子程序间的调用规定一定的规则。ATPCS (ARM Thumb Procedure Call Standard, ARM Thumb 子程序调用标准) 就是 ARM 程序和 Thumb 程序中子程序调用的基本规则。

ATPCS 规定了一些子程序间调用的基本规则。这些基本规则包括子程序调用过程中寄存器的使用规则、数据栈的使用规则、参数的传递规则。

有调用关系的所有子程序必须遵守同一种 ATPCS。编译器或汇编器在 ELF 格式的目标文件中设置相应的属性, 标识用户选定的 ATPCS 类型。对应于不同类型的 ATPCS 规则, 有相应的 C 语言库, 链接器根据用户指定的 ATPCS 类型链接相应的 C 语言库。

使用 ADS 的 C 语言编译器编译的 C 语言子程序满足用户指定的 ATPCS 类型。而对于汇编语言程序来说, 完全要依赖用户来保证各子程序满足选定的 ATPCS 类型。具体来说, 汇编语言子程序必须满足下面 3 个条件:

- ① 在子程序编写时必须遵守相应的 ATPCS 规则。
- ② 数据栈的使用要遵守相应的 ATPCS 规则。
- ③ 在汇编编译器中使用 -apcs 选项。

下面介绍基本 ATPCS。基本 ATPCS 规定了在子程序调用时的一些基本规则, 包括以下 3 方面的内容:

- ① 各寄存器的使用规则及其相应的名称。
- ② 数据栈的使用规则。
- ③ 参数传递的规则。

2. 寄存器的使用规则

寄存器的使用必须满足以下规则。

- ① 子程序间通过寄存器 R0~R3 来传递参数。这时, 寄存器 R0~R3 可以记作 A0~A3。被调用

的子程序在返回前无需恢复寄存器 R0~R3 的内容。

② 在子程序中，使用寄存器 R4~R11 来保存局部变量（编译器会首先将能够映射到寄存器的局部变量分配相应的寄存器，对于不能用寄存器映射的局部变量，比如数组、结构体等多字节局部变量则分配到相应的调用栈中）。这时，寄存器 R4~R11 可以记作 V1~V8。如果在被调用的子程序需要使用 V1~V8 中的某些寄存器来映射本子程序的局部变量，在该子程序进入时必须首先将需要使用的这些寄存器的值压栈，在返回前则必须出栈恢复这些寄存器的值；对于子程序中没有用到的寄存器则不必进行这些操作。在 Thumb 程序中，通常只能使用寄存器 R4~R7 来保存局部变量。

③ 寄存器 R12 用作子程序间的 scratch 寄存器，记作 ip。在子程序间的连接代码段中常有这种使用规则。

④ 寄存器 R13 用作堆栈指针，记作 sp。在子程序中寄存器 R13 不能用作其他用途，寄存器 sp 进入子程序的值和退出子程序的值必须相等。

⑤ 寄存器 R14 称为连接寄存器，记作 lr。它用作保存子程序的返回地址。如果在被调子程序中通过压栈保存了返回地址，寄存器 R14 则可以用作其他用途。

⑥ 寄存器 R15 是程序计数器，记作 PC。它不能用作其他的用途。

表 4-15 总结了在 ATPCS 中各寄存器的使用规则及其名称。这些名称在 ARM 编译器和汇编器中都是预定义的。

表 4-15 寄存器的使用规则

寄存器	别名	特殊名称	使用规则
R15		pc	程序计数器
R14		Lr	连接寄存器
R13		Sp	数据栈指针
R12		Ip	子程序内部调用的 scratch 寄存器
R11	V8		ARM 状态局部变量寄存器 8
R10	V7	Sl	ARM 状态局部变量寄存器 7 在支持数据检查的 ATPCS 中为数据栈限制指针
R9	V6	Sb	ARM 状态局部变量寄存器 6 在支持 RWPI 的 ATPCS 中为静态基址寄存器
R8	V5		ARM 状态局部变量寄存器 5
R7	V4	Wr	ARM 状态局部变量寄存器 4 Thumb 状态工作寄存器
R6	V3		局部变量寄存器 3
R5	V2		局部变量寄存器 2
R4	V1		局部变量寄存器 1
R3	A4		参数/结果/scratch 寄存器 4
R2	A3		参数/结果/scratch 寄存器 3
R1	A2		参数/结果/scratch 寄存器 2
R0	A1		参数/结果/scratch 寄存器 1

3. 堆栈使用规则

栈指针通常可以指向不同的位置。当栈指针指向栈顶元素（即最后一个入栈的数据元素）时，称为 FULL 栈；当栈指针指向与栈顶元素（即最后一个入栈的数据元素）相邻的一个可用数据单元时，称为 EMPTY 栈。

数据栈的增长方向也可以不同。当数据栈向内存地址减小的方向增长时，称为 DESCENDING 栈；当数据栈向内存地址增加的方向增长时，称为 ASCENDING 栈。

综合这两种特点可以有以下 4 种数据栈。

- FD FULL Descending, 满递减栈。
- ED Empty Descending, 空递减栈。
- FA Full Ascending, 满递增栈。
- EA Empty Ascending, 空递增栈。

ATPCS 规定数据栈为 FD 类型, 并且对数据栈的操作是 4 字节对齐的。下面给一个数据栈的示例 (如图 4-33 所示) 及其相关的名词。

- ① 数据栈指针 (stack pointer) 是指最后一个写入栈的数据的内存地址。
- ② 数据栈的基地址 (stack base) 是指数据栈的最高地址。由于 ATPCS 中数据栈是 FD 类型的, 实际上数据栈中最早入栈的数据占据的内存单元是基地址的下一个内存单元。
- ③ 数据栈界限 (stack limit) 是指数据栈中可以使用的最低的内存单元的地址。
- ④ 已占用的数据栈 (used stack) 是指数据栈的基地址和数据栈栈指针之间的区域。其中包括数据栈栈指针对应的内存单元, 但不包括数据栈的基地址对应的内存单元。
- ⑤ 未占用的数据栈 (unused stack) 是指数据栈指针和数据栈界限之间的区域。其中包括数据栈界限对应的内存单元, 但不包括数据栈栈指针对应的内存单元。
- ⑥ 数据栈中的数据帧 (stack frames) 是指在数据栈中, 为子程序分配的用来保存寄存器和局部变量的区域。

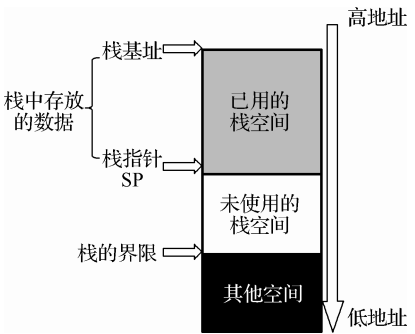


图 4-33 一个数据栈的示意图

4. 参数传递规则

对于参数个数可变的子程序, 当参数不超过 4 个时, 可以使用寄存器 R0~R3 来传递参数; 当参数超过 4 个时, 还可以使用数据栈来传递参数。

```
Int sum5(int a, int b, int c, int d, int e)
{return (a+b+c+d+e);
}

//汇编调用 C 编写的子程序:
AREA sample, CODE, READONLY
IMPORT sum5                                //声明外部标号 sum5, 即 C 函数 sum5 ()
CALLSUM
STMFD SP!, {LR};                          //LR 寄存器入栈
LDR R0, =0x1;                              //设置 sum5 函数入口参数, R0 为参数 a
LDR R1, =0x2;                              //设置 sum5 函数入口参数, R1 为参数 b
LDR R2, =0x3;                              //设置 sum5 函数入口参数, R2 为参数 c
LDR R3, =0x4;                              //设置 sum5 函数入口参数, R3 为参数 d
STR R4, [SP, #-4];                          //参数 e 要通过堆栈传递
BL sum5;                                    //调用 sum5 (), 结果保存在 R0
ADD SP, SP, #4;                             //修正 SP 指针
LDMFD SP, PC;                               //子程序返回
END
```

在参数传递时，将所有参数看作是存放在连续的内存单元中的字数据。然后，依次将各字数据传送到寄存器 R0、R1、R2、R3 中，如果参数多于 4 个，将剩余的字数数据传送到数据栈中，入栈的顺序与参数顺序相反，即最后一个参数先入栈。

按照上面的规则，一个浮点参数可以通过寄存器传递，也可以通过数据栈传递。

子程序中结果返回的规则如下：

结果为一个 32 位的整数时，可以通过 R0 寄存器返回。

结果为一个 64 位整数时，可以通过寄存器 R0 和 R1 返回，依次类推。

结果为一个浮点数时，可以通过浮点运算的寄存器 f0、d0 或 s0 来返回。

结果为复合型的浮点数（如复数）时，可以通过寄存器 f0~fN 或 d0~dN 来返回。

对于为数更多的结果，需要通过内存来传递（此时需要通过 R0 传递指向该内存区域的指针）。

5. C 与汇编的混合编程

在嵌入式系统开发中，目前使用的主要编程语言是 C 和汇编。在稍大规模的嵌入式软件中，例如含有嵌入式操作系统，大部分的代码都是用 C 编写的，主要是因为 C 语言的结构比较好，便于人们理解，而且有大量的支持库。尽管如此，很多地方还是要用到汇编语言，例如开机时硬件系统的初始化，包括 CPU 状态的设定、中断的使能、主频的设定以及 RAM 的控制参数及初始化。一些中断处理方面也可能涉及汇编。另外一个使用汇编的地方就是一些对性能非常敏感的代码块，而要手工编写汇编，达到优化的目的（最好的编译器生成的机器指令与精心编写的汇编代码之间可能依然存在 20%~30% 的性能差距）。

在需要将 C 程序与汇编混和编程时，若汇编代码较简单，则可以直接使用内嵌汇编的方法混和编程；否则，可以将汇编文件以文件的形式加入项目中，通过 ATPCS 规定与 C 程序相互调用、访问。

这里主要讨论 C 和汇编的混合编程，包括相互之间的函数调用。下面分四种情况来进行讨论。

6. 在 C 语言中加入汇编程序

这里介绍在 C/C++ 里加入汇编程序的两种方法：内联汇编（Inline Assemble）和嵌入式汇编（Embedded Assemble）。

内联汇编是指在 C/C++ 函数定义中插入汇编语句的方法，如下面的例子：

```
void enable_IRQ(void)
{
    int tmp;
    __asm                                     //内联汇编定义
    {
        MRS tmp,CPSR                         //可以引用外部的 C 变量定义
        BIC tmp,tmp,#0x08
        MSR CPSR_c,tmp
    }
}
```

内联汇编的用法跟真实汇编之间有很大的区别，并且不支持 Thumb，在内联汇编中不能直接访问物理寄存器（CPSR 除外），即使使用寄存器名进行编程，也会被编译器进行重新分配。

与内联汇编不同，嵌入式汇编具有真实汇编的所有特性，同时支持 ARM 和 Thumb，但是不能直接引用 C/C++ 的变量定义，数据交换必须通过 ATPCS 进行。嵌入式汇编在形式上表现为独立定义的函数体，如下所示：

```

_asm int add(int i,int j)          //定义嵌入式汇编
{
    ADD R0,R0,R1                  //Value of i in R0 and j in R1,result in R0
    MOV PC,LR
}
void main()
{
    Printf("12345+67890=%d\n",add(12345,67890));
}

```

灵活使用内联汇编和嵌入式汇编，可以帮助提高程序效率。

7. 在汇编中使用 C 定义的全局变量

内嵌汇编不用单独编辑汇编语言文件，因此比较简洁，但是有诸多限制，所以当汇编的代码较多时一般放在单独的汇编文件中。这时就需要在汇编和 C 语言之间进行一些数据的传递，最简便的办法就是使用全局变量。

可以在 C 语言函数体外声明一个全局变量，并在需要用到该变量的汇编语言文件中用关键字 **IMPORT** 声明该变量即可，见参数传递规则中的举例。

8. 在 C 语言中调用汇编的函数

在 C 语言中调用汇编文件中的函数，要做的主要工作有两个：一是在 C 语言中声明函数原型，并加 **extern** 关键字；二是在汇编中用 **EXPORT** 导出函数名，并用该函数名作为汇编代码段的标识，最后用 **mov pc, lr** 返回。然后，就可以在 C 语言中使用该函数了。从 C 语言的角度，并不知道该函数的实现是用 C 语言还是汇编。更深层的原因是因为 C 语言的函数名起到表明函数代码起始地址的作用，这和汇编的 **label** 是一致的。

C 语言和汇编之间的参数传递是通过 **ATPCS**（ARM Thumb Procedure Call Standard）的规定来进行的。简单地说就是如果函数有不多于 4 个参数，对应地用 **R0~R3** 来进行传递，多于 4 个时借助栈，函数的返回值通过 **R0** 来返回。

在 C 语言文件中使用汇编语言文件中定义的函数，需要用关键字 **extern** 声明该函数名（即汇编语言中的标号），然后就可以像调用普通的 C 语言函数一样调用该函数了。

```

;汇编函数文件 strcpy.s
;把 R1 指向的数据块复制到 R0 指向的存储块
AREA strcpy, CODE, READONLY
EXPORT strcpy
strcpy
LDRB R2, [R1], #1
STRB R2, [R0], #1
CMP R2, #0
BNE strcpy
MOV PC, LR
END

;C 文件 main.c
;调用 strcpy 函数
extern void strcpy(char *d, const char *s);    //声明 strcpy 为外部引用符号

```

```
main() {  
    char *dest, *source;  
    ;  
    strcpy(dest, source);  
    ;  
}
```

9. 在汇编中调用 C 语言的函数

在汇编中调用 C 语言的函数，需要在汇编中 IMPORT 对应的 C 语言函数名。

在汇编中调用 C 语言的函数，参数的传递也是通过 ATPCS 来实现的。需要指出的是当函数的参数个数大于 4 时，要借助堆栈（stack），具体见 ATPCS 规范。

见参数传递规则中的举例。

4.2.7* ARM 处理器的多核技术

1. 嵌入式多核技术概述

伴随着人们对于个人电脑的性能需求不断提高和处理器制造工艺的进步，桌面处理器推出了同一块芯片内存在多个 CPU 的技术方案。而移动处理器很长时间内都使用了单核方案，直到近年来，多核 CPU 的移动设备才逐渐开始普及。一方面，人们对于设备的性能、散热、体积提出了更高的要求；另一方面，以 Android 和 iOS 为代表的移动操作系统也日益盛行，这些操作系统下用户可以实现各种不同的功能，处理不同的事物；此外，包括娱乐休闲、办公商务、监控安全、交通运输等领域也在发展的过程中逐渐转向高性能智能化设备。以上种种原因为高性能多核移动处理器的发展提供了契机。移动设备处理器从最初单核，发展到双核、四核乃至八核，处理器的工作频率也从最初的数百兆赫到现在最高的接近 2GHz。移动处理器的多核化仍在不断深入的过程中，为了应对更复杂的应用情形甚至还出现了同时存在不同计算能力内核（big.LITTLE）的架构（4.2.8 节中将介绍 big.LITTLE 架构）。而在目前的各种多核处理器架构中，SMP（对称多处理）架构依然是最基本的也是应用最广泛的架构之一。

SMP 系统中多个 CPU 的并行处理为性能的提升带来了拓展空间，但是 CPU 数目的增加并不会达到相应比例的性能提升。而关于处理器数目与性能提升之间的关系，由 IBM 雇员 Gene Amdahl 提出了一个以他的名字命名的法则，用以预测最大的预期系统改进。它主要用来计算使用多处理器后理论上的最大性能改进。该法则以函数的形式表现为：

$$\text{Speedup} = \frac{1}{F + (1 - F) / N}$$

上式用来计算系统的最大性能改进，式中 N 表示处理器的数目，而因数 F 指系统程序中不能并行化的系统部分，即本质上顺序执行的系统部分。根据阿姆达尔定律，可以得到图 4-34，图中显示了在不同并行比例条件下不同处理器数量对于性能增加的曲线。其实我们很容易理解这个定律，试想如果一个程序的可并行部分只占到整个程序的 50%，那就意味着即使我们可以通过多个处理器并行计算这部分工作，将这部分的工作时间降为零（如果我们采用的处理器足够多的话），我们也只是消除了本程序一半的工作量，因此对于这个程序，最极端的优化也只能达到单核系统的两倍。该定律告诉我们，并不是并行处理的内核越多越好，多核加速的关键其实最终取决于工作任务本身的可并行性。

ARM 公司早在 2004 年就发布了基于 ARM v6 架构 ARM11 的多处理器实现 ARM11 MPCore，可最多集成 4 个 ARM11 内核。近年来，ARM 公司又接连发布了 ARM Cortex-A9 MPCore、ARM Cortex-A7 MPCore 和 ARM Cortex-A15 MPCore 等一系列多处理器架构方案。随着性能、功耗、联网等一系列要

求,多核处理器已经不再是简单的对称多处理结构了,更多的异构多核处理器架构开始出现,例如 ARM 公司就有 big.LITTLE 架构,但是对称多处理技术仍然是众多多核架构的基础。

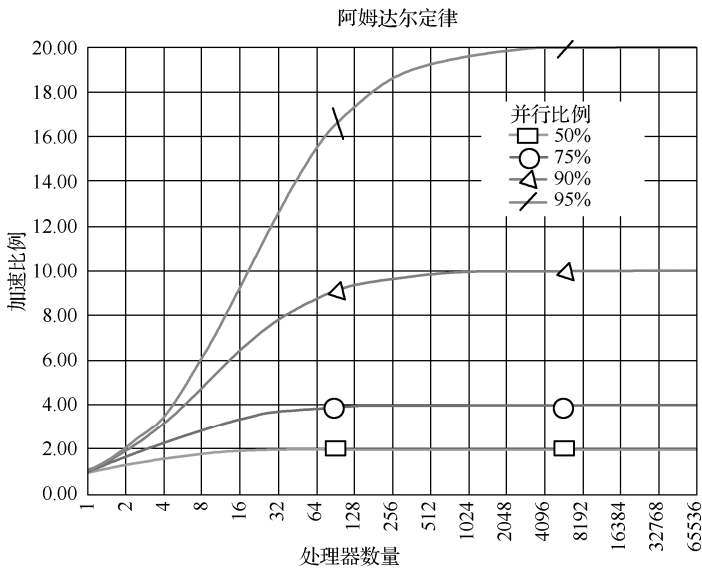


图 4-34 阿姆达尔定律

2. 处理器的连接

对称多处理系统是一个紧耦合的多处理器系统,它由一组相同的处理器组成,每个处理器都独立运行,但是处理器之间可以共享内存、I/O 设备、中断系统的资源。处理器之间由总线、跨接开关或者 mesh 网络所连接。通常每个处理器都有与之相关的私有高速缓存来加快处理器与内存之间的数据读取速度,降低系统总线需求。

ARM 公司从 ARM11 处理器系列开始推出 ARM11MPCore 系列多核处理器产品,结构可见图4-35,其可以实现拓展最多 4 颗 ARM v6k 内核。ARM11 MPCore 中 SCU 模块负责保持 L1 数据缓存的一致性,分布中断控制器在完成一般中断控制器功能的同时还需完成中断的分配。与单核 ARM v6 不同的是,ARM11 MPCore 对于 ARM v6 做出了一系列拓展,包括:

- 指令的加入,如 WaitForInterrupt 指令,该指令对于在 ARM 多核处理器的启动过程中十分关键。
- 对于一些寄存器的保留位的使用和作用的修改,如 CP15 中的控制寄存器 c1 的 TEX 位(位: 12~14)、C 位(位: 3)和 B 位(位: 2)。这些位原本是内存可缓存性的控制位,但在实际应用中无法同时全部用到这些位所提供的选项。于是只是用 TEX[0], C 和 B 位用于控制选择,余下的两个位则可以被用作操作系统管理的页表位。

ARM Cortex-A9 MPCore 基于基础 ARM v7 架构的多处理拓展解决方案,见图 4-36,同 ARM11 MPCore 类似,它也具有 SCU、GIC(全局中断控制器)这样的硬件单元来保持内存一致性和中断控制,它还提供了 ACP 接口,该接口可用作标准的 AMBA 3 AXI 辅助接口,它支持所有标准读写事务,而对连接的组件没有任何其他一致性要求。不过,针对一致的内存区域的任何读事务都会与 SCU 交互,以测试所需信息是否已存储在处理器的 L1 缓存内。如果存储在其中,则会将其直接返回到请求组件。如果未存储在 L1 缓存中,则在最后转发到主内存之前还有机会存储在 L2 缓存中。对于针对任何一致的内存区域的写事务,在将写入数据转发到内存系统之前,SCU 会强制其保持一致性。此外,此事务还可分配到 L2 缓存,从而消除直接写入对芯片外内存产生的功耗和性能影响。Cortex-A9

MPCore 对于 ARM v7 的扩展被称为 Multiprocessing Extensions。这些扩展的主要作用是增强多处理中的缓存以及分支预测的作用。新增 PLDW 指令与原有的 PLD 指令功能完全相同，同样是数据的预加载，但是当内存类型为共享时，PLDW 指令引起的 linefill 将会造成其他处理器内的数据失效，从而使得可以向该行缓存写入数据。因为多核在同时运行操作系统时每个内核都拥有自己的虚拟内存地址，这就需要多个 TLB（Translation Lookaside Buffer）的存在，因而 Cortex-A9 MPCore 在 VMSA（Virtual Memory System Architecture）架构中还增强了 TLB 操作。

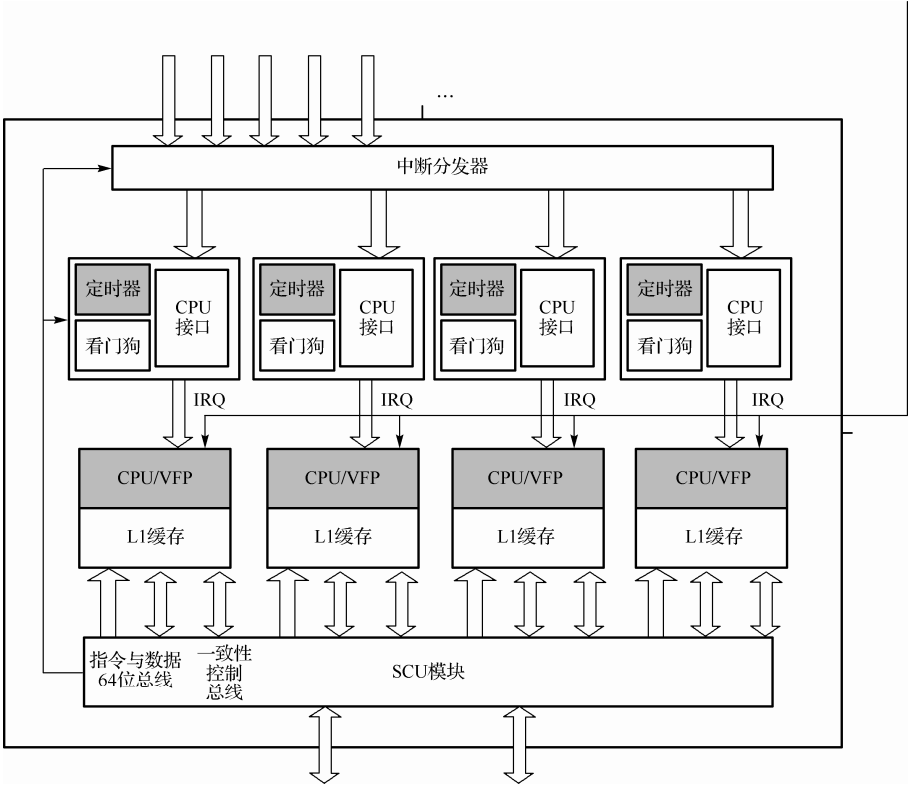


图 4-35 ARM11 MPCore 结构示意图

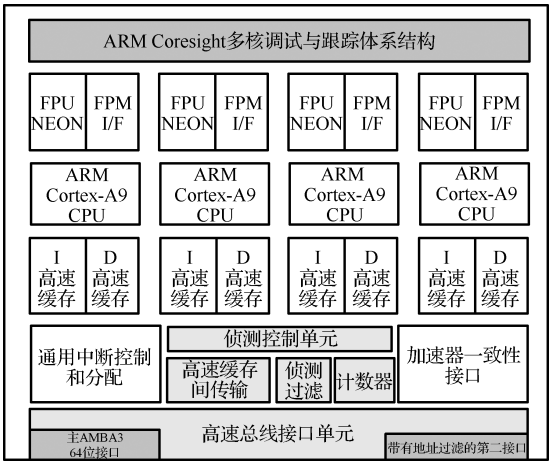


图 4-36 Cortex-A9 MPCore 结构示意图

3. 缓存 (Cache) 一致性

为了加快多核处理器内各个 CPU 访问数据,每个 CPU 都会有私有的高速缓存 (Cache) 来暂存刚刚访问的数据和指令。然而,在同一个系统中存在多个 CPU,每个 CPU 都可以独立执行任务,不可避免地会访问同一块内存区域。若任一 CPU 对相关区域的数据进行了变更,则其他 CPU 缓存所存储的该区域的数据就与实际内存中不一致,如果仍旧按照缓存中存储的数据运行的话,所产生的结果就可能产生错误。避免缓存不一致带来的系统性能损耗十分重要。

在 OMAP 4460 中,每个 Cortex-A9 内核都自带 32KB 的指令缓存和数据缓存,两个处理器内核共享 1MB 的 L2 缓存。在 ARM v7 架构之前,架构定义的缓存控制机制只能覆盖到 L1 缓存,而对于其他层级的缓存支持是根据具体实现来定义的。ARM v7 结构的处理器可以通过 CP15 协处理器的 c0 寄存器获取不同的缓存信息,具体获取的信息根据 MRC 指令下不同的 opcode1 和 opcode2 组合的不同而不同。当 opcode1 为 0 且 opcode2 为 1 时,c0 代表 CTR (Cache Type Register, 缓存类型寄存器),可以从中知道缓存索引 (index) 和标记 (tagging) 的方式以及最小缓存行大小。当 opcode1=0, opcode2=1 时,c0 代表 CLIR (Cache Level ID Register, 缓存层 ID 寄存器),可以获得缓存的级数 (如 L1 和 L2)。

在基本的 ARM v7 架构中,处理器进行缓存维护操作的影响范围只是该处理器本身。而在多处理器环境下,缓存维护操作不仅需要对发出该操作的处理器进行操作,往往还必须通过 IPI (Inter Processor Interrupt) 的方式通知其他处理器执行同样的缓存操作。上文提过的针对 TLB 的拓展也是多处理器环境下对缓存的影响所做的应对措施。

为了解决多核情况下带来的缓存一致性问题,ARM11 MPCore 系列处理器中出现了一个模块,即 SCU (Snoop Control Unit),它的功能有:

- 保持数据缓存在核间的一致性。
- 获得 L2 AXI 内存访问。
- 仲裁核间 L2 缓存访问申请。
- 管理 ACP (Accelerator Coherency Port, 加速器一致性端口) 访问。

SCU 模块可以被认为是 ARM 多核的核心技术。SCU 通过监视各个处理器内核的缓存存取数据的地址,如果有一个写入操作发生在已被缓存的内存地址上,缓存控制器就会把已被缓存在其他 CPU 中的相同 Cache 行设为无效,从而避免了内存不一致的情况,此过程可参见图 4-37。其中,CPU0 对其缓存的一行地址进行了写操作 (图中阴影部分),SCU 一方面将 CPU0 所写的内容写穿到下一级缓存,同时扫描其他 CPU 核的 Cache 中是否也缓存了相同的行,如果发现 CPU2 的缓存中也保留了该地址内容的一个副本,则必须将该行内容标注为无效 (Invalidate)。此后即使 CPU2 发起了对该地址的访问,被标注为无效的行也将造成本次访问的缺失,并强制使 CPU2 的 L1 Cache 发起对该行的重新填充。对于由于这种一致性问题而造成的 Cache 缺失,我们将其称为一致性缺失 (Coherence Miss)。

ACP 模块主要用于连接原本不被处理器缓存管理的 AXI Master 外设,例如 DMA Engine,示意图可参见图 4-38。以往的设计中,如果有一块内存会被硬件 DMA 直接更新数据,则该块内存就会被设定为 non-Cached (不被缓存的),以避免因缓存该部分内存,而其他硬件也可以修改此段内存的内容 (比如 DMA),导致两者不一致而造成冲突。non-Cached 技术的缺点是处理器必须等待较慢的数据存取,导致性能的下降。通过 SCU 上的 ACP 模块,就可以让硬件 DMA 更改数据的同时与处理器内部缓存保持一致。ACP 主要目的是为了让其他设备也可以共享并存取 L1/L2 缓存中的数据,以期可以在不增加系统功耗的情况下 (减少到外部存储器存取的次数) 提高系统性能。

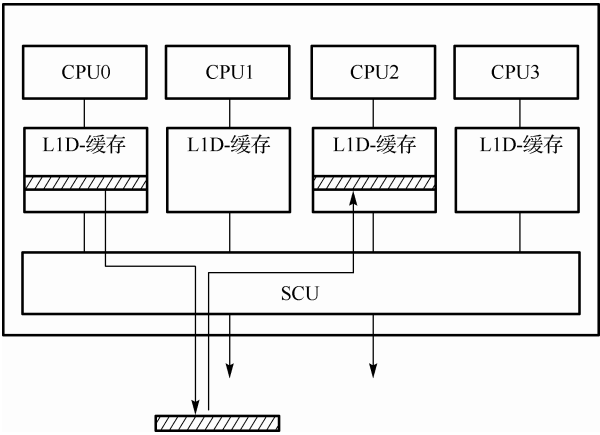


图 4-37 SCU 工作示意图

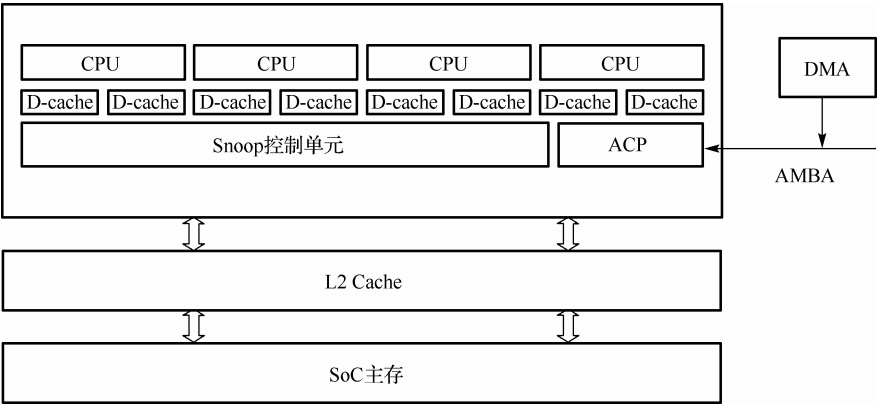


图 4-38 ACP 模块示意图

4. 多核中断技术

Cortex-A9 MPCore 中的中断控制器为 GIC (Genetic Interrupt Controller)，它具有标准化、结构化的特点，因此可以在各个 ARM 系列处理器中看到它的存在。它负责处理所有与中断控制器相连接的中断资源。GIC 架构在逻辑上分为中断分配块 (Distributor block) 和 CPU 接口块，见图 4-39。中断分配块负责中断优先级的排定和向各个 CPU 接口块分发中断。而每个 CPU 接口模块除了能够接收到分发下来的中断，还需要能够实现中断的优先级屏蔽和中断的抢占处理。

多核情况下的中断来源与单核处理器几乎相同，一般的 SGI (Software Generated Interrupt)、全局定时器、FIQ、IRQ 和共享的外设中断对于各个 CPU 来说是共享的，都需要通过中断分配块分发到具体的 CPU 上，唯一不同的是 MPCore 架构上每个 CPU 都有自己独立的定时器和看门狗，它们的中断都是只需要所属 CPU 进行响应的。

另外，ARM 的 MPCore 系列采用了 1-N 中断处理模型：在双核或者更多核的系统中，如果同一个中断请求被超过一个内核所接收到，MPCore 确保有且只有一个处理器会读取相应中断 ID 并做出反应。使用这个模型就减少了中断服务例程中一次锁的需求。还有一种中断处理模型为 N-N，该模型中每个处理器对中断的处理行为相互独立，即使有处理器对中断进行了清除，而这个操作不会影响到其他处理器上的中断状态。

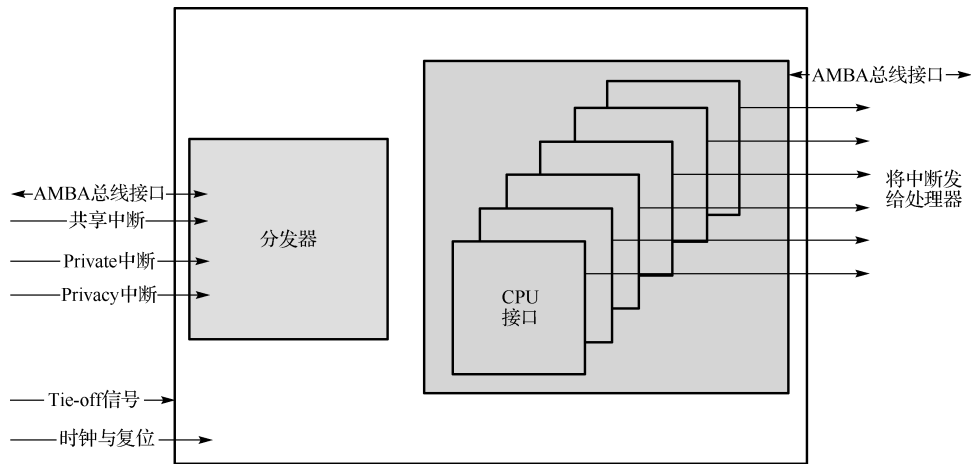


图 4-39 GIC 中断分发模块和 CPU 模块图

GIC 的中断过程如图 4-40 所示，开始于从中断接口模块处接收到中断，根据中断的优先级和处理器
的状态（中断开启与否、屏蔽指定中断）利用 **Prioritization and Selection**（优先级排定与选择）功能
找到优先级最高的中断并将该中断通过 CPU 接口块分发给指定的处理器。GIC 会帮助每个处理器维护
一个尚未处理的中断列表，并且选择优先级最高的中断发给对应的处理器，如果中断优先级相同，则
选择中断号较低的中断进行触发。中断列表中的信息包括中断号和中断优先级。当收到处理器发出的
EOI 信号（End of Interrupt Information）时，确认对应正在处理的中断已被处理完毕，或是处于正处理
的状态，GIC 就会改变所维护的中断列表状态。MPCore 处理器的中断可以处于以下 3 种状态。

- **Inactive**：该中断可能尚未被触发或是已被触发并且在该处理器中已被处理完毕。同时，该中
断源还可能在其他处理器中还处在 **Pending** 或是 **Active** 的状态，会根据每个处理器处理中断的
情况而定。
- **Pending**：该中断已发生，但尚未在对应处理器上触发执行。
- **Active**：该中断已经被执行，但尚未执行结束。

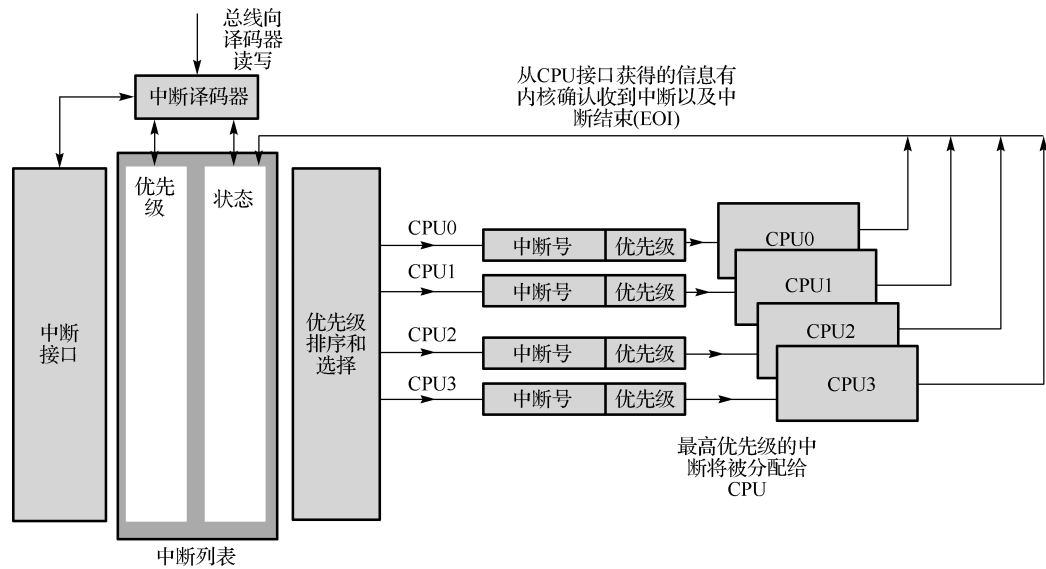


图 4-40 多核系统下的中断各模块工作流程图

当中断管理模块检测到中断发生时，就会设定对应目标处理器该中断的状态为 Pending。如果该中断为电平触发，有任意 MPCore 处理器对该中断的状态还处于 Active 时，该中断就不能被设定为 Pending。如果是边缘触发，当前一个中断尚未处理完毕，下一个中断又发生时，在 MPCore 中，对不同处理器可能同时存在 Pending 和 Active 状态。

MPCore 每个处理器的 CPU Interface 可支持中断优先级屏蔽和中断抢占。一个处于 Pending 状态的中断如果通过优先级屏蔽，并且优先级高于当前处理器正在执行中的 Active 中断，就会被该处理器插入执行。当处理器通过 IAR（Interrupt Acknowledge Register）寄存器读取目前要处理的中断号时，CPU Interface 就会记录该中断的优先级并通知 GIC 将该中断标识为 Active。如果在处理器读取 IAR 寄存器前，该中断因为优先级屏蔽的更改或是通过 IPCR（Interrupt Pending Clear Register）寄存器被取消了，就会从 IAR 寄存器读取到 1023，表示没有需要被处理的中断。当中断处理完毕时，就会需要处理器设定 End of Interrupt Register，用以通过 CPU Interface 通知 GIC 将该中断标识为 Inactive。各状态之间的转换过程示意图见图 4-41。

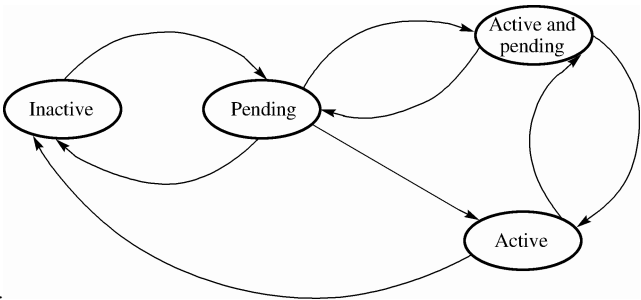


图 4-41 中断处理状态转换图

4.2.8* ARM 处理器的最新发展

ARM v7 架构有 3 种处理器系列。

① 系列 A（ARM v7-A）：需要运行复杂应用程序的“应用处理器”。支持大型嵌入式操作系统（不一定实时），比如 Linux、Android、iOS 以及微软的 Windows CE 和智能手机操作系统 Windows Mobile。这些应用需要强大的处理性能，并且需要硬件 MMU 实现的完整而强大的虚拟内存机制，还基本上会配有 Java 支持，有时还要求一个安全程序执行环境（用于电子商务）。典型产品包括高端手机和手持仪器、电子钱包以及金融事务处理器。

② 系列 R（ARM v7-R）：硬实时且高性能的处理器。目标是高端实时市场。像高档轿车的组件、大型发电机控制器、机器人手臂控制器等，它们使用的处理器不但要很好很强大，还要极其可靠，对事件的反应也要极其敏捷。

③ 系列 M（ARM v7-M）：面向传统单片机的应用而量身定制。在这些应用中，尤其是对于实时控制系统，低成本、低功耗、极速中断反应以及高处理效率都是至关重要的。

Cortex 系列是 v7 架构的第一次亮相。下面将重点介绍 ARM v7 A 系列处理器。另外，我们在本书的 5.2.6 节和 5.3.6 节中将分别介绍 Cortex A8 处理器的 Cache 结构和 Cortex A9 处理器的虚拟存储管理技术。

1. ARM Cortex A9

Cortex-A9 是基于指令集 ARM v7 的 A 系列处理器，许多主流处理器应用对性能的要求都日益提高，以实现更快的数据速率、更多的媒体服务和新功能。在应用领域，既要求低成本又要求高性能的

实例不在少数，比如，联网笔记本电脑及其他便携式设备、手机、PDA、机顶盒应用、游戏机以及车载信息娱乐设备等。采用多核处理器架构不但能够解决峰值性能的要求，而且其设计也能够大大降低功耗。多核设备具有性能可扩展性高和功耗低的特点，为设计提供了极大的灵活性。ARM Cortex-A9 处理器是 ARM 处理器系列中一款高性能处理器产品，该款处理器采用了广受支持的 ARM v7 架构。Cortex-A9 处理器的设计是基于推测型九到十二级流水线（取决于不同的功能单元），该流水线具有高效、动态长度、多发射超标量及乱序执行特征。Cortex-A9 微架构提供两种选项：可扩展的 Cortex-A9 MPCore 多核处理器或较为传统的 Cortex-A9 单核处理器。可扩展的多核处理器和单核处理器（两款不同的独立产品）支持 16KB、32KB 或 64KB 四路组关联一级缓存的配置。两款产品采用了相同的微架构，整合了多种特色功能，使处理器内核和整个集成系统的架构功能、性能及功效得到了大幅提升。单核处理器比 ARM11 级设备提供了更好的性能和功效，不但增强了移动设计的功能，而且降低了功耗水平，延长了电池使用寿命。而在实现方面，这款处理器还具有出色的架构软件兼容性，能够在达到 Cortex-A8 级性能的前提下降低成本，从而扩大了相关软件投资的市场应用范围。而 MPCore 型处理器则拥有先进的电源管理功能，能够进一步降低功耗，达到并超过了日益增多的市场和应用对功耗的要求。

Cortex-A9 单核处理器拥有首屈一指的性能和功效，对于要求高性能的低功耗、成本敏感、基于单核处理器的设备，它无疑是理想的解决方案。Cortex-A9 处理器采用了一种便利的可综合 IP（软核），为 ARM11 提供了理想的升级通道。Cortex-A9 单核处理器为独立指令和数据传输提供两个低延时 Harvard64-bit AMBA3 AXITMMaster 接口，在通过内存缓存区复制数据时，每 5 个处理器周期能维持 4 次双字写入。Cortex-A9 处理器的框图如图 4-42 所示。

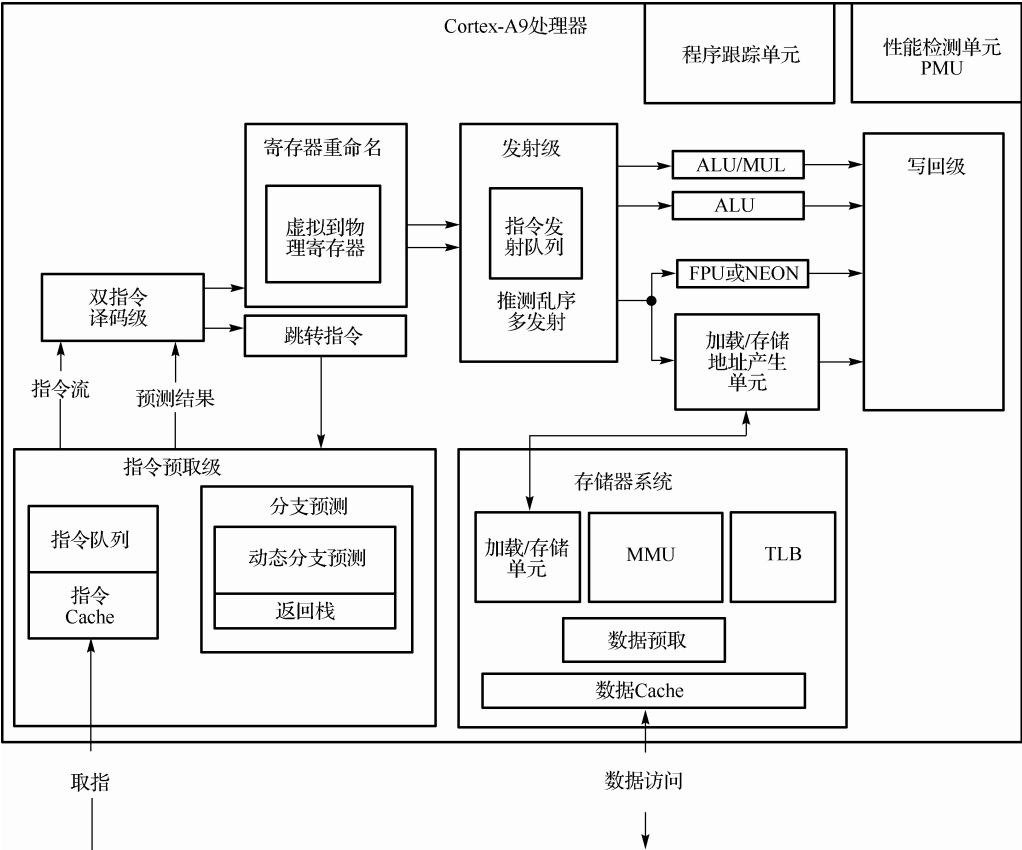


图 4-42 Cortex-A9 单核处理器的框图

Cortex-A9 MPCore 多核处理器集成了大获成功的 ARM MPCore 技术，Cortex-A9 MPCore 不仅能够空前提升峰值性能，同时有效地支持了设计灵活性和新的功能，从而进一步降低了控制处理器及系统层次的功耗。利用 ARM MPCore 技术的设计灵活性和先进的功耗管理技术，Cortex-A9 MPCore 的针对性应用能够在有限的功耗下维持移动设备的正常运转，从而为移动设备带来优于现有解决方案的峰值性能。这种处理器充分利用了可扩展峰值性能，在性能上超越了现有的同等高端嵌入式设备，并在更为广阔的市场中维持了持续稳定的软件投资。

Cortex-A9 处理器为包括手机、高端消费类电子和企业产品在内的多种市场应用提供了一种具有可扩展性的解决方案，因为该款处理器满足了以下各项要求：

- 降低了功耗、提升了性能；
- 提升峰值性能，适应各种要求最为严苛的应用；
- 开发不同设备时可复用软件和工具；
- 两款 Cortex-A9 处理器以及 Cortex-A9 NEON 皆具有完美的应用兼容性；
- 媒体处理引擎（MPE）或浮点运算单元（FPU），还能加强特定应用中的性能表现，进一步扩大了这两款处理器的市场应用范围。

两款处理器的应用设计配置均具有极大的灵活性，允许根据具体应用和特定市场的要求进行定制，如表 4-16 所示。

表 4-16 ARM Cortex-A9 处理器特点

特点	优势
高效超标量流水线	性能独占鳌头，超过 2.0 DMIPS/MHz，实现了前所未有的峰值性能，同时维持了较低的功耗，延长了电池寿命，降低了封装和操作成本
NEON 媒体处理引擎	加快媒体和信号处理，提升了具体应用性能，更有应用软件开发和支持相统一的便利
浮点运算单元	显著提高了单精度及双精度标量浮点运算的速度。性能较原有的 ARM FPU 提升了一倍，提供了行业领先的图像处理、图形和科学运算能力
优化的一级缓存	优化了一级缓存的性能和功耗，结合了最低存取延时技术，不但最大限度地提升了性能，而且将功耗降至最低。同时还为更强大的处理器相互通信提供了高速缓存一致性；能够有力支持具有丰富 SMP 功能的操作系统，从而简化多核软件开发
Thumb-2 技术	性能上能达到传统 ARM 代码的峰值水平，最多可将指令存储所需内存减少 30%
TrustZone 技术	为包括数字版权管理和电子支付在内的安全应用提供了可靠的实施方案。广受技术及行业合作伙伴的支持
Jazelle RCT 和 DBX 技术	最多可将字节码语言的实时（JIT）和预先编译代码大小减少 3 倍，同时支持直接执行 Java 指令的字节代码，提高了传统虚拟机的速度
二级缓存控制器	在高频设计或者需要降低片外内存存取功耗的设计中，能维持较低的延时和较高的带宽存取，最高可配置 2MB 的缓存内存
程序跟踪宏单元和 CoreSight 设计套件	这两个组件的结合使软件开发者能够轻松地跟踪多个处理器的执行历史并将其与带有时间标记的相关系数一起存储在片内缓冲器之中或通过标准跟踪接口传到芯片外面，从而增强了开发和调试的可视性

Cortex-A9 流水线介绍：

- 先进的取指及分支预测处理——避免因访问指令的延时而影响跳转指令的执行。
- 最多支持 4 条指令 Cache 行预取挂起（prefetch-pending）——进一步减少了内存延时的影响，从而促进指令的顺利传输。
- 每个周期内可连续将 2~4 条指令发送到指令解码——确保充分利用超标量流水线性能。
- Fast-loop 模式——执行小循环时提供低功耗运行。
- 超标量解码器——每个周期可完成两条指令的完全解码。
- 支持指令预测执行——通过将物理寄存器动态地重新命名至虚拟寄存器池来实现。

- 提升了流水线的利用效率——消除了相邻指令之间的数据依赖性，减少了中断延时。
- 支持寄存器的虚拟重命名——以一种有效的、基于硬件的循环展开方法，提高了代码执行效率，而不会增加代码大小和功耗水平。
- 4 个后续流水线（subsequent pipeline）中的任何一个均可从发射队列中选择执行指令——提供了无序分配，进一步提高了流水线利用效率，无需借助于开发者或编译器指令调度。确保专为上一代处理器进行优化的代码能够发挥最大性能，也维护了现有软件投资。
- 每周期支持两个算术流水线（full dual arithmetic pipeline）、加载-存储（load-store）或计算引擎以及分支跳转的并行执行。
- 可将有相关性 load-store 指令提前传送至内存系统进行快速处理，进一步减少了流水线暂停，大幅提高了涉及存取复杂数据结构或 C++ 函数的高级代码的执行效率。
- 支持 4 个数据 Cache Line 的填充请求——而且还能通过自动或用户控制预取操作，保证了关键数据的可用性，从而进一步减少了内存延时导致的暂停现象。
- 支持无序指令完成回写（write back）——允许释放流水线资源，无需受限于系统提供所需数据的顺序。

2. ARM Cortex A7 和 A15

如何在保证较长续航时间的前提下实现更高的计算性能一直是困扰智能手机的难题，可以设计出具有强大性能的处理芯片，但是却很难实现长时间续航，而如果要为了节能推出低功耗芯片，那么芯片的计算能力又不能满足应用的要求。在这种矛盾中，移动芯片市场上的做法是将高性能计算芯片跟低功耗计算芯片联合。

其实自 A9 处理器开始，ARM 就开始在乱序执行内核方面发力（在改进的并行计算中，指令的执行将根据任务进行重新排序）。ARM Cortex A15 高性能内核即是继续延续 A9 的步伐，而 A7 则是走了相反的一条路——回归到 Cortex A8 的顺序执行，在并行计算中可以顺序执行两条指令，不过 A7 跟原来的 A8 还是有着很多不同的。A8 的设计可以追溯到 2003 年，后来 ARM 计划推出合成多内核的版本，用来提升频率和性能，不过受制于将新产品推向市场的时间和研发上的成本压力，“加强版”的 A8 内核计划夭折。如今得益于市场、工艺等，ARM 具有了卷土重来的资本，A7 内核可以帮助 ARM 在未来几年内奠定在高频率、低功耗芯片方面的优势。

Cortex A7 具有 8 级流水线，并能支持指令双发射。与 A8 不同的是，A7 不支持双发浮点或者说 NEON 指令集。在内部结构上的很多方面，A7 都跟 A8 相似，不过在 FPU 等方面得到大幅加强。尽管限制了双发能力，ARM 希望 A7 能提供比 A8 更强的每时钟周期性能和整体性能，由于采用了比 A8 更先进的预测器，A7 的分支预测计算能力得到提升，更好的预测算法也使得这颗芯片更为节能，此外在 A7 中采用了更低延迟的 L2 缓存（每次访问为 10 时钟周期），不过具体的情况还要取决于芯片制造厂商的工艺。

限制带宽的设计让 A7 的芯片面积可以做到很小，28nm 的单核 Cortex A7 的面积仅有 0.5mm^2 ，在相同的工艺上，ARM 希望合作厂商能将 A7 的硅片面积控制在 A8 的 1/2 甚至 1/3，而 A9 的硅片面积跟 A8 差不多。与之相对的是，高性能的 A15 要比两者大得多。ARM Cortex A 系列处理器架构的比较表参见表 4-17。

Cortex A7 与 Cortex A15 实现 100% 的 ISA 兼容，而且 A7 能够支持新的虚拟化指令集、支持整数除法和 40-bit 内存寻址。也就是说任何运行在 A15 内核上的代码都可以在 A7 上运行，只是运算速度要慢一些。不过这一点让 SoC 芯片同时搭载 Cortex A15 和 Cortex A7 具有了实际意义，二者也可以根据任务负载的不同及时即时切换，ARM 把这种机制称为“big.LITTLE”技术，如图 4-43 所示。

表 4-17 各种架构比较

	处理器					
	Cortex-A5	Cortex-A7	Cortex-A8	Cortex-A9	Cortex-A12	Cortex-A15
发布时间	2009.12	2011.10	2006.7	2008.3	2013.6	2011.4
典型时钟频率	~1GHz	28nm 下~1GHz	65nm 下~1GHz	40nm 下~2GHz	28nm 下~2GHz	28nm 下~2.5GHz
执行顺序	顺序执行	顺序执行	顺序执行	乱序执行	乱序执行	乱序执行
内核数量	1~4 个	1~4 个	1	1~4 个	1~4 个	1~4 个
峰值整数吞吐量	1.6DMIPS/ MHz	1.9DMIPS/ MHz	2DMIPS/ MHz	2.5DMIPS/ MHz	3.0DMIPS/ MHz	3.5DMIPS/ MHz
VFP 结构	VFP v4	VFP v4	VFP v3	VFP v3	VFP v4	VFP v4
NEON 结构	NEON	NEON v2	NEON	NEON	NEON v2	NEON v2
半精度扩展	有	有	无	有	有	有
硬件除法指令	无	有	无	无	有	有
FMA 操作	有	有	无	无	有	有
流水线级数	8	8	13	9~12	11	15+
每周期译码的指令数	1	部分指令并发	2（超标量）	2（超标量）	2（超标量）	3（超标量）
返回堆栈数	4	8	8	8	8	48
LPAE	无	有	无	无	有	有
浮点单元	可选	有	有	可选	有	可选
AMBA 接口	64 位 AMBA 3	128 位 AMBA4	64 或 128 位 AMBA 3	2*64 位 AMBA 3	128 位 AMBA 4	128 位 AMBA 4
通用中断控制器	有	可选	无	有	外部	可选
跟踪	可选的 ETM	可选的 ETM 单独宏单元	集成的 ETM	集成的 PTM	集成的 PTM	集成的 PTM

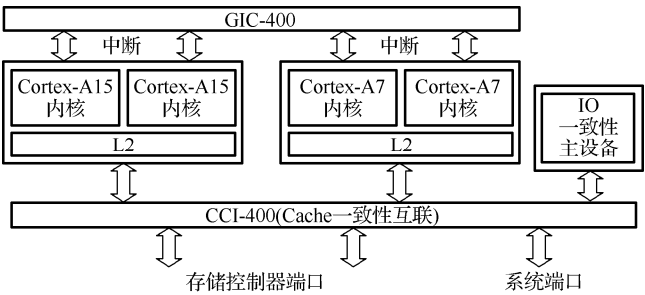


图 4-43 big.LITTLE 大小核架构

实际上，我们可以把 Cortex A15 理解为 ARM 架构在运算能力上的重大提升。Cortex A15 面向智能手机和平板机市场，最终将取代 Cortex A9。A15 要比 A9 更为强大，在高负载下，Cortex A15 将可能比 A9 更为节能——A15 内核在执行任务时尽管需要比 A9 更多的功耗，不过在完成任务后转入休眠状态的切换过程要比 A9 短得多，因此 A15 芯片的平均能耗将比 A9 低不少。

ARM 认为：乱序执行的微处理器架构在执行对运算能力要求并不高的移动设备任务时效率非常低。Cortex A15 作为乱序执行架构的一员也自然未能“免俗”，因此 ARM 推出了低功耗的 A7 跟它协作——当智能手机在你口袋中处于待机状态时，只有 A7 内核在运行。

尽管 Cortex A7 内核也可以单独运行（意为离开了 A15 也能单独做成芯片），但是 ARM 合作厂商可能更倾向于将 A7 跟 A15 集成在一起，也就是所谓的 big.LITTLE 方案。

由于 A7 跟 A15 内核均能支持完全相同的 ARM 指令集，所以任何应用都可以在两种内核上运行，在一颗 SoC 芯片内同时具有 A15 内核和 A7 内核，不过在操作系统看来，这颗 SoC 只是拥有两颗可以运行相

同指令集的内核——并没有把二者区分开来。区分二者的是 ARM 的能源管理固件——它可以根据任务负载的不同激活不同的内核。此外，高速缓存一致性的设计也让两种内核之间的任务切换更为平滑——**两种内核之间的切换延迟仅为 20ms**。A7 和 A15 的性能和功耗比较见图 4-44。因此在 big.LITTLE 架构中，任何时候系统要么运行在 A7 处理器上，要么运行在 A15 处理器上，但不会同时运行在两者上。

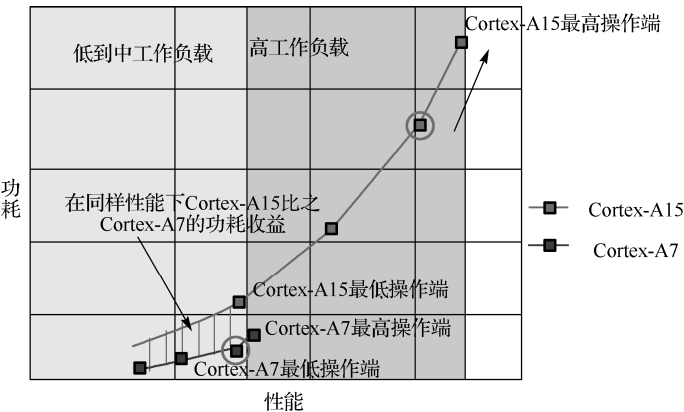


图 4-44 Cortex A7 和 A15 的性能和功耗比较

4.3* 其他 CPU 介绍

ARM 处理器已经成为移动智能终端类应用的事实标准，几乎所有的智能手机和平板电脑都采用基于 ARM 处理器的 SoC 作为核心的应用处理器。与此同时，ARM 公司还在不遗余力地进军其他应用领域，一方面 Cortex M 系列处理器在不断蚕食传统的 8 位与 16 位单片机市场，另一方面最高端的 Cortex A 系列，尤其是其多核处理器性能已接近传统的服务器处理器，使得 ARM 公司进军服务器市场的意图也越来越明显。所有的迹象似乎都在表明 ARM 似乎想垄断所有的处理器市场。然而，处理器市场，尤其是在嵌入式应用领域，很难真正形成类似 Intel 在桌面市场形成的垄断。因此，作为嵌入式系统的入门教材，我们有必要向读者介绍其他的 CPU 架构。

我们将在本节介绍两款非 ARM 的处理器架构：MIPS 和 UniCore。MIPS 架构是最早期的 RISC 架构，其设计完美而纯粹地体现了 RISC 的设计哲学，而其在不同领域的卓越表现（至少是在 ARM 出现前的那段时间里）也值得我们在本书中为其留有一席之地。我们介绍 MIPS 的另一个原因是因为作为国产 CPU 代表之一的龙芯处理器就是在 MIPS 架构上发展而来的。另外，国内的君正公司推出的 XBurst 内核也采用了兼容 MIPS 指令集的架构。

UniCore-2（众志）是北京大学微处理器研究中心推出的完全自主知识产权的 CPU，从 2000 年前后，众志团队一直致力于高性能处理器的研究与设计。作者所在的国家专用集成电路系统工程技术研究中心曾经先后在 2002 年和 2009 年采用众志内核设计面向移动智能应用的 SoC 设计。众志内核采用完全自主的指令集设计，这虽然给处理器设计者最大的设计自由，但也给编译器团队和构建基于该处理器的生态环境造成了很大的困难。

显然，嵌入式领域还存在其他大量的优秀架构，比如 Sparc、Power PC，甚至 Intel 的 x86 等等。受限于篇幅与作者知识的局限，我们仅在本书中介绍两款与国产 CPU 相关的架构。在当前 CPU 核基本被国外厂商垄断的大环境下，不管是站在国家信息安全的角度，还是站在国内产业安全的角度，研发推广自主 CPU 架构无疑具有国家战略高度的意义。然而，一款 CPU 架构从技术成熟（包括性能、功耗和相应的工具链、操作系统等）走向产业成熟需要走过太多的艰难险阻。且不说在一些技术细节

上（比如能效比），我们与国外最先进的处理器依然存在一些差距，更大的问题是如何构建基于自主处理器架构的软件生态环境。（也许 Android 是一个福音，但大量的 Android 应用基于 NDK 的事实又成为国产处理器绕不过去的坎。）无论如何，不管是龙芯还是众志，作为设计自主 CPU 的先驱团队，作者都要向他们表达崇高的敬意。

4.3.1 MIPS 体系架构

早在 1980—1984 年间 UC Berkeley 的 David Patterson 就提出了 RISC 的概念，并设计出了相应的处理器。也许是历史的巧合，几乎在同时（1981—1984 年），就在离 UC Berkeley 不远的 Stanford 大学，John L. Hennessy^①领导的一个项目组正在设计被称为 MIPS 的另外一款新处理器。UC Berkeley 的 RISC 处理器取得了巨大的成功，以至于后续所有采用类似设计思想与理念的处理器都被称为 RISC 处理器，当然也包括 MIPS 处理器。这两款诞生于硅谷名校的处理器对于后续的处理器的设计都产生了深远的影响，RISC 的商业化版本就是大名鼎鼎的 SPARC 处理器，甚至现在如日中天的 ARM 处理器架构也深受其影响。而 MIPS 架构除了自己推出了一系列商业化版本之外，还对 DEC 公司里程碑式的 Alpha 处理器和惠普公司的 Precision 处理器，甚至国产 CPU 龙芯 Loongson 以及君正 XBurst 都产生了巨大的影响。然而，虽然都被称为 RISC 处理器，从诞生之日起，这两款处理器乃至后续受其影响的处理器就秉承着不同的 RISC 设计哲学与理念。

在 RISC 体系结构中，MIPS 也许是最优雅的一种，甚至连竞争对手也不得不承认这一点。MIPS 自身优雅的设计虽然不能在充满竞争的市场中保证长盛不衰，但其微处理器却总能跻身于每一代最有效率的微处理器之列并保持最简洁的设计。与 ARM 机构相比这种简洁性体现在以下几方面：

① 与 ARM 处理器至少 7 种工作模式相比，MIPS 架构通常只有 3 种工作模式（User、Supervisor、Kenel）。

② MIPS 架构没有 CPSR 这样的寄存器，也不包括诸如溢出、进位、负数和零标志位，其条件跳转只取决于寄存器比较的结果。

③ MIPS 架构将内存管理、Cache 甚至中断与异常处理等事务统统交给 CP0 协处理器。

④ MIPS 架构有 32 个通用寄存器，HI 和 LO 两个专用寄存器，而且在所有的工作模式下都是软件可见的，而没有采用所谓寄存器窗口或寄存器 Bank 的概念（SPARC 处理器采用寄存器窗口加速子函数调用，ARM 则在不同的模式下保留特定的 r13、r14 等寄存器）。

⑤ 与 ARM 相比，MIPS 的寻址方式要简单得多，只有寄存器基址加立即数偏移的方式，没有 ARM 的前变址、带回写前变址、后变址以及立即数偏移、寄存器偏移、寄存器加移位偏移等复杂的寻址方式，也没有 ARM 处理器的批量加载与存储指令。

1. MIPS 架构的指令集

MIPS 处理器的整数指令设计非常简单，总的来说 MIPS 将所有整数指令分为 3 类：寄存器型指令（R）、立即数型指令（I）和跳转型指令（J）。所有的指令都是 32 位的，其中寄存器类指令在 32 位编码中，操作码占用了 6 比特，第一源寄存器和第二源寄存器分别占用 5 比特编码（MIPS 有 32 个通用寄存器），目的寄存器也占用 5 比特编码。在寄存器类指令的低 11 位分别是 5 比特的移位量和 6 比特的功能编码。立即数类型的指令由 6 比特操作码和两个 5 比特的寄存器编码加 16 比特的立即数构成。跳转类指令则由 6 比特操作码加 26 比特地址偏移量构成。表 4-18 总结了这 3 类指令的格式。

① 对！你没看错，他就是那本大名鼎鼎的 *Computer Architecture: A Quantitative Approach, Fifth Edition* 《计算机体系结构量化研究方法（第 5 版）》的作者，这本书也是我们首先推荐的扩展阅读。

表 4-18 MIPS 处理器的三类指令格式

类型	指令格式（比特）					
R	Opcode（6）	rs（5）	rt（5）	rd（5）	Shamt（5）	Funct（6）
I	Opcode（6）	rs（5）	rt（5）	立即数（16）		
J	Opcode（6）	地址（26）				

MIPS 的核心指令，包括整数算术逻辑指令、存储访问指令及分支指令。这些指令的实现过程大致相同，而与具体的指令类型无关。实现每条指令的前两步是一样的：

- 程序计数器（PC）指向指令所在的存储单元，并从中取出指令。
- 通过指令字段内容，选择读取一个或两个寄存器。对于取字指令，只需读取一个寄存器，而其他大多数指令要求读取两个寄存器。

这两步之后，为完成指令而进行的步骤则取决于具体的指令类型。幸运的是，对三种指令类型（存储访问、算术逻辑、分支）的每一种而言，其动作大致相同，与具体指令无关。MIPS 指令集的简洁和规整使许多指令的执行很相似，因而简化了实现过程。

例如，所有指令在读取寄存器后都要使用算术逻辑单元（ALU）。存储访问指令用 ALU 计算地址，算术逻辑指令用 ALU 执行运算，分支指令用 ALU 进行比较。在使用 ALU 之后，完成不同指令所需的动作就有所不同了。存储访问指令需要访问内存以便读取和存储数据。算术逻辑指令和装载指令将来自 ALU 或存储器的数据写入寄存器。对分支指令，我们需要基于比较的结果决定是否改变下一条指令地址；如果不修改下一条指令地址，则下一条指令地址默认是当前指令地址+4。

MIPS 处理器有 32 个通用整数寄存器，所有的数据必须先暂存到这些寄存器后才能够对其进行算术运算。寄存器\$0 的值永远都为常数 0，而寄存器\$1 则通常保留给汇编器使用（用于处理伪指令和大量数）。表 4-19 给出了 MIPS 整数指令，表 4-20 给出了 MIPS 的浮点指令。

表 4-19 MIPS 处理器的整数指令

分类	指令名	指令语法	含义	类型	备注
算术指令	加	add \$d,\$s,\$t	\$d = \$s + \$t	R	两寄存器相加，如溢出，触发异常
	无符号数加	addu \$d,\$s,\$t	\$d = \$s + \$t	R	同上，但忽略溢出
	减	sub \$d,\$s,\$t	\$d = \$s - \$t	R	两寄存器相减，如溢出触发异常
	无符号数减	subu \$d,\$s,\$t	\$d = \$s - \$t	R	同上，但忽略溢出
	加立即数	addi \$t,\$s,C	\$t = \$s + C（有符号）	I	寄存器加有符号立即数，如溢出，触发异常
	加无符号立即数	addiu \$t,\$s,C	\$t = \$s + C（无符号）	I	同上，但忽略溢出
	乘法	mult \$s,\$t	LO = ((\$s * \$t) << 32) >> 32; HI = (\$s * \$t) >> 32;	R	两寄存器相乘，64 位结果写入指定寄存器：LO 和 HI
	无符号乘	multu \$s,\$t	LO = ((\$s * \$t) << 32) >> 32; HI = (\$s * \$t) >> 32;	R	两寄存器相乘，64 位结果写入指定寄存器：LO 和 HI
	除法	div \$s, \$t	LO = \$s / \$t HI = \$s % \$t	R	两寄存器相除，商存放在 LO，余数存放在 HI
	无符号除	divu \$s, \$t	LO = \$s / \$t HI = \$s % \$t	R	两寄存器相除，商存放在 LO，余数存放在 HI
数据传送	装载字	lw \$t,C (\$s)	\$t = Memory[\$s + C]	I	从地址 MEM[\$s+C]装载一个字到\$t
	装载半字	lh \$t,C (\$s)	\$t = Memory[\$s + C]（有符号）	I	从地址 MEM[\$s+C]装载一个半字到\$t，且符号位扩展

续表

分类	指令名	指令语法	含义	类型	备注
数据 传送	装载无符号半字	lhu \$t,C (\$s)	\$t = Memory[\$s + C]（无符号）	I	从地址 MEM[\$s+C]装载一个半字到\$t
	装载字节	lb \$t,C (\$s)	\$t = Memory[\$s + C]（有符号）	I	从地址 MEM[\$s+C]装载一个字节到\$t，且符号位扩展
	装载无符号字节	lbu \$t,C (\$s)	\$t = Memory[\$s + C]（无符号）	I	从地址 MEM[\$s+C]装载一个字节到\$t
	存储字	sw \$t,C (\$s)	Memory[\$s + C] = \$t	I	将\$t 存入 MEM[\$s+C]
	存储半字	sh \$t,C (\$s)	Memory[\$s + C] = \$t	I	将\$t 低 16 位存入 MEM[\$s+C]
	存储字节	sb \$t,C (\$s)	Memory[\$s + C] = \$t	I	将\$t 低 8 位存入 MEM[\$s+C]
	加载立即数到高位	lui \$t,C	\$t = C << 16	I	将 16 位立即数装载到\$t 的高 16 位
	从 HI 装载到寄存器	mfhi \$d	\$d = HI	R	将寄存器 HI 的内容装载到\$d，该指令前后两条不能是乘法或除法指令
	从 LO 装载到寄存器	mflo \$d	\$d = LO	R	将寄存器 LO 的内容装载到\$d，该指令前后两条不能是乘法或除法指令
	从控制寄存器装载	mfcZ \$t, \$d	\$t = Cp[Z].ControlRegister[\$d]	R	从协处理器 Z 的控制器装载到通用寄存器
	存储控制寄存器	mtcZ \$t, \$d	Cp[Z].ControlRegister[\$d] = \$t	R	将通用寄存器的值写入协处理器 Z 的控制寄存器
逻辑 运算	与	and \$d,\$s,\$t	\$d = \$s & \$t	R	位与
	与立即数	andi \$t,\$s,C	\$t = \$s & C	I	高 16 位填充 0
	或	or \$d,\$s,\$t	\$d = \$s \$t	R	位或
	或立即数	ori \$t,\$s,C	\$t = \$s C	I	高 16 位填充 0
	异或	xor \$d,\$s,\$t	\$d = \$s ^ \$t	R	按位异或
	或非	nor \$d,\$s,\$t	\$d = ~ (\$s \$t)	R	按位或非
	小于则置位	slt \$d,\$s,\$t	\$d = (\$s < \$t)	R	测试两个寄存器的大小
	小于立即数则置位	slti \$t,\$s,C	\$t = (\$s < C)	I	测试寄存器和立即数的大小
移位 指令	逻辑左移立即数	sll \$d,\$t,\$amt	\$d = \$t << \$amt	R	寄存器逻辑左移 \$amt 位
	逻辑右移立即数	srl \$d,\$t,\$amt	\$d = \$t >> \$amt	R	寄存器逻辑右移 \$amt 位
	算术右移立即数	sra \$d,\$t,\$amt		R	寄存器算术右移 \$amt 位
	逻辑左移	sllv \$d,\$t,\$s	\$d = \$t << \$s	R	寄存器逻辑左移\$s 位
	逻辑右移	srlv \$d,\$t,\$s	\$d = \$t >> \$s	R	寄存器逻辑右移\$s 位
	算术右移	srav \$d,\$t,\$s		R	寄存器算术右移\$s 位
条件 分支	相等则跳转	beq \$s,\$t,C	if (\$s == \$t) go to PC+4+4*C	I	若两个寄存器相等则跳转到相应地址
	不等则跳转	bne \$s,\$t,C	if (\$s != \$t) go to PC+4+4*C	I	若两个寄存器不等则跳转到相应地址
绝对 跳转	绝对跳转	j C	PC = PC+4[31:28] . C*4	J	绝对跳转
	绝对寄存器跳转	jr \$s	goto address \$s	R	跳转到寄存器\$s 指定的地址
	跳转并连接	jal C	\$31 = PC + 4;PC = PC+4[31:28] . C*4	J	用于函数调用，返回地址保存在寄存器\$31

表 4-20 MIPS 处理器浮点指令

分类	指令名	指令语法	含义	类型	备注
算术指令	单精度加	add.s \$x,\$y,\$z	\$x = \$y + \$z		
	单精度减	sub.s \$x,\$y,\$z	\$x = \$y - \$z		
	单精度乘	mul.s \$x,\$y,\$z	\$x = \$y * \$z		
	单精度除	div.s \$x,\$y,\$z	\$x = \$y / \$z		
	双精度加	add.d \$x,\$y,\$z	\$x = \$y + \$z		
	双精度减	sub.d \$x,\$y,\$	\$x = \$y - \$z		
	双精度乘	mul.d \$x,\$y,\$z	\$x = \$y * \$z		
	双精度除	div.d \$x,\$y,\$z	\$x = \$y / \$z		
	协处理器装载字	lwcZ \$x,CONST (\$y)	Coprocessor[Z].DataRegister[\$x] = Memory[\$y + CONST]		
	协处理器存储字	swcZ \$x,CONST (\$y)	Memory[\$y + CONST] = Coprocessor[Z].DataRegister[\$x]		
比较指令	单精度比较 (eq, ne, lt, le, gt, ge)	c.lt.s \$f2,\$f4	if (\$f2 < \$f4) cond=1;else cond=0		
	双精度比较 (eq, ne, lt, le, gt, ge)	c.lt.d \$f2,\$f4	if (\$f2 < \$f4) cond=1;else cond=0		
跳转指令	浮点比较为真时跳转	bc1t 100	if (cond == 1) go to PC+4+100		
	浮点比较为假时跳转	bc1f 100	if (cond == 0) go to PC+4+100		

2. MIPS 指令集与 ARM 相比较的特点

ARM 与 MIPS 处理器在同年发布并遵循相同的简洁的设计哲学。表 4-21 列出了 ARM 与 MIPS 的相似性，二者的主要区别是 MIPS 有更多的寄存器而 ARM 有更多的寻址模式。表 4-22 展示了 MIPS 与 ARM 在算术逻辑和数据传输指令方面具有相似的指令集核。对于比较和条件分支，ARM 使用传统的存储在程序状态字中的 4 位条件码来决定条件分支是否执行，而 MIPS 使用寄存器中的值来决定条件分支是否执行。

表 4-21 ARM 和 MIPS 指令集的相同点

	ARM	MIPS
发布时间	1985	1985
指令大小 (位)	32	32
寻址空间 (大小、模式)	32 位，平坦	32 位，平坦
数据对齐	对齐	对齐
数据寻址模式	9	3
整数寄存器 (个数，模式，大小)	15 个通用寄存器×32 位	31 个通用寄存器×32 位
I/O	存储器映射	存储器映射

3. MIPS 架构的未来

对 MIPS 公司而言，相对简洁的设计是一种商业上的需要，该公司起源于 1985 年一个制造并销售芯片的学术项目。后来，该体系结构得到了（或许现在仍继续得到）工业界制造商们最广泛的支持——从生产专用集成电路核心（ASIC core）的厂商，如 MIPS Technologies、飞利浦，到制造低成本 CPU 的

厂商，如 IDT、AMD/Alchemy，再到广泛应用的嵌入式领域中唯一的 64 位 CPU 厂商，如 PMC-Sierra、东芝和 Broadcom。

MIPS 架构的发展如图 4-45 所示。

表 4-22 ARM 的寄存器-寄存器指令和数据传输指令和 MIPS 是等价的

	指令名	ARM	MIPS
寄存器-寄存器	加法	add	addu, addiu
	加法（溢出捕获）	adds	add
	减法	sub	subu
	减法（溢出捕获）	subs	sub
	乘法	mul	mult, multu
	除法	—	div, divu
	与	and	and
	或	orr	or
	异或	eor	xor
	取寄存器高位	—	lui
	逻辑左移	lsl	sllv, sll
	逻辑右移	lsr	srlv, srl
	算术右移	asr	srav, sra
	比较	cmp, cmn, tst, teq	slt/i, slt/iu
数据传输	取有符号字节	ldrsb	lb
	取无符号字节	ldrb	lbu
	取有符号半字	ldrsh	lh
	取无符号半字	ldrh	lhu
	取字	ldr	lw
	存字节	strb	sb
	存半字	strh	sh
	存字	str	sw
	读、写特殊寄存器	mrs, msr	move
	原子交换	swp, swpb	ll, sc

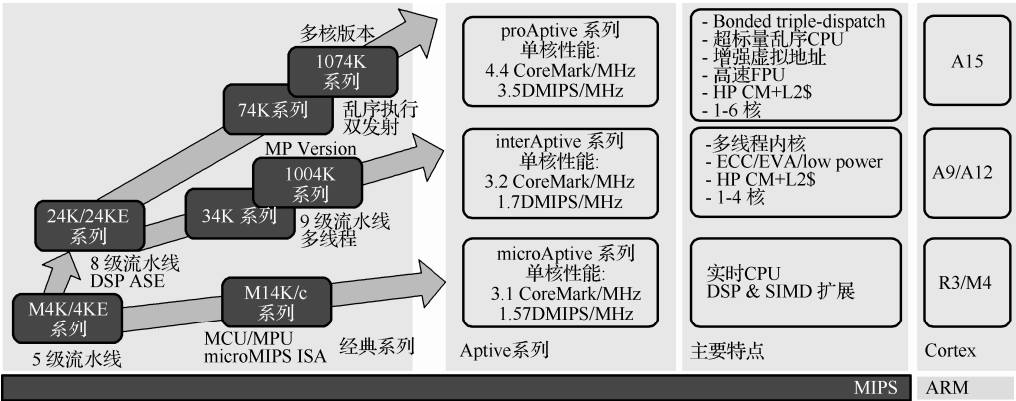


图 4-45 MIPS 架构的发展

4.3.2 龙芯处理器

龙芯处理器(英文名 Loongson, 早期英文名曾经是 Godson)是由中科院计算所设计的 MIPS64 指令集兼容的通用处理器家族。虽然目前的龙芯处理器完全兼容 MIPS64 指令集,但其内部的微架构设计却是完全独立自主完成的。早期的龙芯处理器为了规避 MIPS 公司的专利,其指令集中缺少 4 条 MIPS 指令,因此早期的龙芯处理器只能被称为 MIPS-Like 的处理器。2007 年,中科院计算所与 MIPS 公司达成协议,由 ST 公司(STMicroelectronics 公司)为龙芯购买 MIPS 授权。2009 年,中科院计算所直接向 MIPS 公司授权了 MIPS32 和 MIPS64 架构。2011 年龙芯公司向 MIPS 公司授权了 MIPS32 和 MIPS64 架构以进一步设计和推广龙芯处理器。从 2002 年开始,龙芯团队不断推出新的处理器产品,形成了龙芯 1 号、龙芯 2 号和龙芯 3 号三代处理器家族。

龙芯 1 号(英文名称 Godson-1)于 2002 年研发完成,是一颗 32 位的处理器,主频 266 MHz,架构类似于 MIPS R1X000。2005 年,龙芯在微软授权和帮助下,完成了 BSP 支持包的设计,并通过了微软的 NSTL 内部测试,从而成为第一款事实上全面支持 WindowsCE 系统的国产通用处理器。2006 年 3 月,龙芯通过购买意法半导体授权 MIPS64 架构获得 MIPS 指令的使用权,暂时通过购买技术脱离知识产权的法律纠纷。

龙芯 2 号(英文名称 Godson-2)于 2003 年正式完成并发布。龙芯 2 号是 64 位处理器,主频为 300~500MHz。龙芯 2 号最初的版本采用 0.13 μ m 制造工艺。龙芯 2 号其实是一个系列处理器,经过数次演化,已形成 2、2A、2B、2C、2E、2F 等型号。其中 2006 年推出的龙芯 2E 是一款采用 64 位 MIPS III 指令集的 RISC 处理器,扩展了多条多媒体指令,采用 90nm 的 CMOS 工艺,晶体管数目是 4700 万个,芯片面积是 6.8mm \times 5.2mm。最高主频达到 1.0GHz,一般频率是 800MHz,功耗为 5~7W。龙芯 2E 处理器拥有 128KB 一级缓存、512KB 二级缓存。它的单精度浮点运算速度为每秒 80 亿次,双精度浮点运算速度是每秒 40 亿次。

龙芯 3A 处理器是首款中国国产商用 4 核处理器,主要用于低功耗服务器、个人计算机、工业控制、网络安全等领域。主频达到 1GHz,峰值计算能力达到 16GFLOPS,功耗约 15W。频率为 1GHz 时双精度浮点运算速度峰值达到每秒 160 亿次,单精度浮点运算速度峰值每秒 320 亿次。龙芯 3A 采用意法半导体公司 65nm CMOS 工艺生产,晶体管数目达 4.25 亿个。龙芯 3A 集成了四个 64 位超标量处理器核、4MB 的二级 Cache、两个 DDR2/3 内存控制器、两个高性能 HyperTransport 控制器、一个 PCI/PCIX 控制器以及 LPC、SPI、UART、GPIO 等低速 I/O 控制器。龙芯 3A 的指令系统与 MIPS64 兼容并通过指令扩展支持 x86 二进制翻译。龙芯 3B 为 8 核处理器,主频达到 1GHz,支持矢量运算加速,峰值计算能力达到 128GFLOPS,主要用于高性能计算机、高性能服务器、数字信号处理等领域。

本节将以龙芯 2F 为例简单介绍一下龙芯处理器。龙芯 2F 的基本流水线有 9 级:取指、预译码、译码、寄存器重命名、调度、发射、读寄存器、执行、提交等。指令流水线每个时钟周期取 4 条指令进行译码,并且动态地发射到 5 个全流水的功能部件中,这 4 条指令中,定点指令和浮点指令最多两条,访存指令一条。

多发射结构针对指令集的并行性,不仅要求一个周期内发出的多条指令间互不相关,而且要求流水线管道中各条指令间也互不相关。因此龙芯的四发射超标量技术使指令流水线中指令和数据相关问题变得更为复杂,龙芯 2F 采用指令的乱序执行技术和较为激进的存储系统设计来提高流水线的效率,图 4-46 所示是龙芯 2F 的框图,图中 Fix RS 表示定点单元的保留站,Float RS 则是浮点单元的保留站。

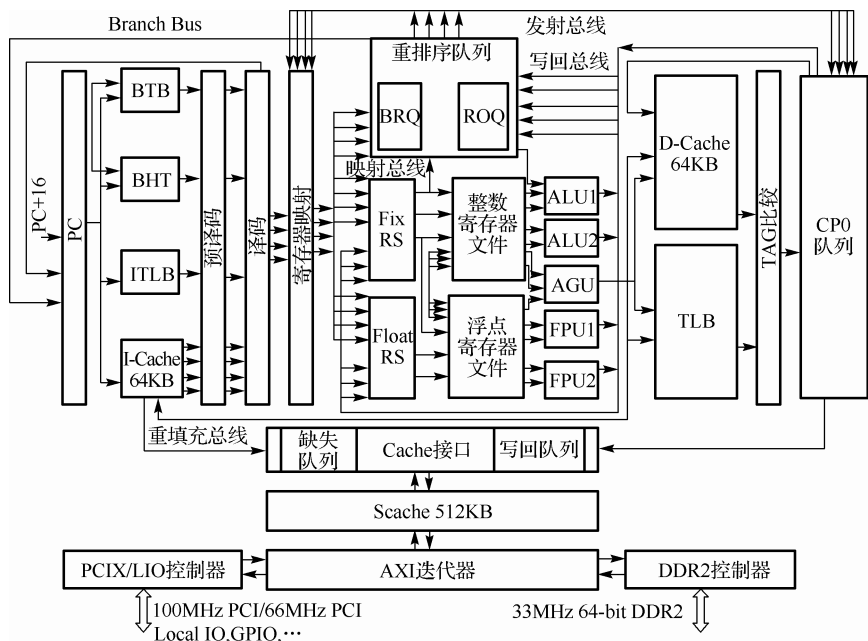


图 4-46 龙芯 2F 处理器框图

1. 乱序执行

乱序执行的目的是提高每一周期内指令执行的数量,让 CPU 尽可能地满负荷运转来提高程序运行的速度。CPU 在保证依赖关系的前提下,根据各单元的空闲状态和各指令能否提前执行的具体情况析后,将能提前的指令立即发送给相应的功能单元执行。各单元不按规定顺序执行完指令,但必须由相应单元再将运算结果重新按原来指定的指令顺序排列后提交,以保证精确中断和访存顺序执行。乱序执行技术包括寄存器重命名、动态调度、转移预测等。

1) 寄存器重命名

指令间的依赖关系阻碍指令调度发挥作用,进而影响指令级并行。寄存器重命名是将特定物理寄存器和结构寄存器建立动态关联的过程,给指令中每个目标寄存器分配一个新的物理寄存器以解决读后写 (WAR) 和写后写 (WAW) 相关,并用于例外和错误转移预测引起的精确现场恢复。龙芯 2F 通过 64 项的物理寄存器堆进行定点和浮点寄存器的重命名。

2) 动态调度

以硬件复杂性的增加为代价,通过硬件重新安排指令的执行顺序,来调整相关指令实际执行时的关系,减少处理器空转。根据指令操作数准备好的次序而不是指令在程序中出现的次序来执行指令,减少了 RAW (写后读) 相关引起的阻塞。龙芯 2F 有一个 16 项的定点保留站和一个 16 项的浮点保留站用于乱序发射,并通过一个 64 项的 Reorder 队列 (ROB) 实现乱序执行的指令按照程序的次序提交。

3) 分支预测

通过预测程序流来调整指令的执行,并分析程序的数据流来选择指令执行的最佳顺序,减少由于控制相关引起的阻塞。龙芯 2F 使用 16 项的转移目标地址缓冲器 (BTB),2K 项的转移历史表 (BHT),每项采用 2 位,9 位的全局历史寄存器 (GHR),以及 4 项的返回地址栈进行分支预测。全局历史寄存器与转移历史表共同构成 GShare 预测器 (关于分支预测器可以参阅 4.1.4 节)。但与其他高性能 CPU 相比,龙芯 2F 处理器的指令分支预测器显得比较简单,比如 BTB 比较小,相比于 Alpha 21264 4096

项的全局预测器,龙芯 2F 的 BHT 的表项数也不是非常多,且不说 21264 还采用了全局预测器与局部预测器混合的 Tournament 预测器。

2. 浮点部件

龙芯 2F 的 FPU 提供一个和 CPU 指令流水线并行的指令流水线。它和 CPU 共享基本的 9 级流水体系结构,但根据浮点操作的不同,执行流水线又细分为 2~6 个流水级。

FPU 由 FALU1、FALU2 两个功能单元组成。FALU1 单元执行浮点加减、浮点乘法、浮点乘加、取绝对值、取反、精度转换、定浮点格式转换、比较、转移等操作,FALU2 执行浮点加减、浮点乘法、浮点乘加、浮点触发、浮点开平方操作。每个 FALU 单元每个周期能够接收一条指令,并能向浮点寄存器文件送出一个结果。

FALU1 单元是全流水的,浮点加减、浮点乘法、浮点乘加运算需要 6 个执行周期;定点与浮点间的格式转换运算需要 4 个执行周期;FALU1 中的其他所有运算需要 2 个执行周期。

FALU2 单元执行浮点加减、浮点乘法、浮点乘加、浮点除法、浮点开平方根操作。其中浮点加减、浮点乘法、浮点乘加操作为全流水操作,需要 6 个执行周期;浮点除法根据操作数的不同,需要 4~16 个执行周期;浮点开平方根操作则需要 4~31 个执行周期。浮点除法和开平方根操作不是流水的。

龙芯 2F 还定义了 32 个 64 位浮点寄存器。

3. Cache 的组织

一级 Cache 由 64KB 的指令 Cache 和 64KB 的数据 Cache 组成,二级 Cache 大小为 512KB,均采用四路组相联的结构、随机替换算法以及写回、非阻塞读策略。

1) Cache 的访问

龙芯 2F 的 Cache 采用四路组相联结构,其中含有 512 个索引项。当对 Cache 索引时,同时访问四个组中的 Tag 和 Data。然后将四个组中的 Tag 与转换后的物理地址部分进行比较,从而确定命中哪一个数据行。

龙芯 2F 的一级 Cache 和二级 Cache 均采用随机替换算法。随机替换方法的实现较为简单,但是 Cache 冲突失效问题较为严重。其他 CPU 中常见的实现方法往往是一级缓存采用 LRU 替换算法,而二级缓存采用随机替换算法。另外,似乎龙芯 2F 的二级缓存的容量也偏小了一些。

2) 非阻塞 Cache

非阻塞 Cache 机制允许一次 Cache 缺失而后面的多个 Cache 访问继续进行,由于不用等待前面缺失的访问完成,使得 MLP(访存级并行)的程度更高,从而提高系统的整体性能。在一个阻塞 Cache 的设计中,当发生某个 Cache 失效时,处理器将暂停后续的访存操作。此时,处理器开始一个存储周期,取出被请求的数据,将其填入 Cache 中,然后再恢复执行。在非阻塞 Cache 设计中,Cache 并不会在某个失效上暂停,如果通过调度将会引起失效的访存指令连续发射,形成访存流水线,则可以有效地重叠访存的延迟,加快程序运行速度。

龙芯 2F 支持多重失效下的命中,它最多可以支持 24 项的访存队列和 8 项的访存失效队列。非阻塞 Cache 的结构能更有效地使用循环展开和软件流水。

4. 内存管理

为了能够快速地进行虚拟地址到物理地址的映射,龙芯 2F 处理器采用了较大的、全关联映射机制的 64 项 TLB。每项可以映射一个奇页和一个偶页,同时维护页面的 Cache 一致性属性。通过 24 项的访存队列以及 8 项的访存失效队列来动态地解决地址依赖,实现访存操作的乱序执行、非阻塞 Cache、取数指令猜测执行、写合并等优化技术。

4.3.3 UniCore-2 处理器

UniCore-2 处理器是北京大学微处理器研究开发中心研制的北大众志 UniCore 系列处理器之一，其特点是高性能、高集成度、低功耗，内部集成了支持 32 位 UniCore32 指令系统的定点微处理器核、支持 IEEE 754 标准的 64 位浮点协处理器以及系统控制模块等。

UniCore-2 可提供接近 ARM1136 的内核处理性能，大约 1.0 Dhrystone MIPS（百万条指令每秒）。UniCore-2 内核采用哈佛结构，可同时访问分离的指令 Cache 和数据 Cache，并可同时访问分离的指令 MMU 和数据 MMU（该处理器采用的是实地址的 Cache，必须先进行地址转换后才能访问 Cache）。UniCore-2 内核采用统一的地址和数据总线，作为和系统中其他部分的接口，该接口符合 AMBA2.0 的 AHB 总线规范，因此，UniCore-2 内核作为一个 AHB 总线主设备连接在整个 SoC 的设计之中（新的众志 CPU 可以支持 AMBA AXI 总线接口）。

UniCore-2 内核基于北大众志自主定义的 UniCore32 指令体系结构，其内核结构如图 4-47 所示。从图中可以看出该 CPU 采用整数内核与浮点内核分开的设计，这可能是因为 UniCore-2 采用顺序单发射的执行机制，无法真正实现整数指令与浮点指令并发执行。另外，UniCore-2 的 TLB 设计采用了二级结构，其 L1 TLB 采用的是全关联设计，而 L2 TLB 则可能采用组关联的方式进行设计，该架构应该是在性能和能耗上的一个折中（与之对应地，龙芯 2F 采用 64 项全关联 TLB 则是向性能最优进行的优化）。

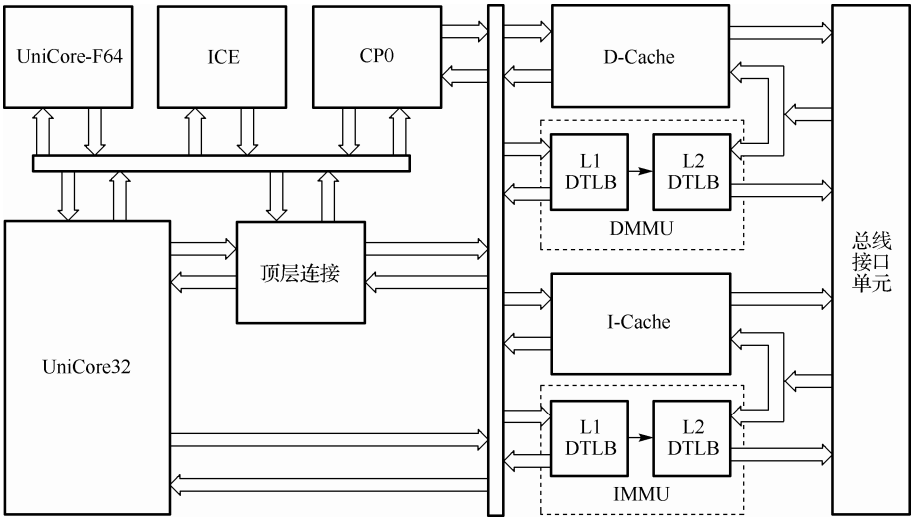


图 4-47 UniCore-2 内核结构框图

1. UniCore32 概述

UniCore-2 内核支持 32 位定点指令系统体系结构 UniCore32 和 64 位浮点指令系统 UniCore-F64。UniCore32 采用 Load/Store 形式的通用寄存器结构。内存地址空间采用 32 位线性寻址，每一个内存单元是一个字节，字长为 32 位，采用小端（Little Endian）存储格式。

定点处理器核采用单发射顺序执行的 8 级流水线结构，提供了乘加指令、多字内存-寄存器转存指令、前导 0/1 处理指令、自增/自减寻址方式等 DSP 增强功能，可高效支持 FFT、FIR 等 DSP 应用典型算法的实现，提供了交换指令、多种模式切换、实时中断机制等支持操作系统的高效运行。

系统控制协处理器 P0、片载调试器 P1 和浮点协处理器 P2 均采用与定点处理器核紧密耦合的协处理器接口。其中,浮点协处理器 UniCore-F64 支持单精度和双精度浮点加法、乘法、除法和求绝对值等浮点运算指令,符合 ANSI/IEEE754-1985 标准。系统控制协处理器 P0 支持 UniCore-2 处理器功能配置,包括使能和禁用 Cache/MMU、控制数据 Cache 行操作模式等。

总的来说,UniCore 2 处理器的架构吸取了 ARM 处理器架构的特点,同时也部分吸取了 MIPS 架构的特点。比如 UniCore 2 的工作模式,寄存器 Bank 和程序状态寄存器的设计具有比较明显的 ARM 特色,而 32 个通用寄存器的设计又比较接近 MIPS 架构。采用 32 个通用寄存器的设计使得三操作数指令的寄存器编码由 ARM 的 12 位(每个寄存器占 4 比特)增加到 15 位(每个寄存器占 5 比特),这使得 UniCore 的指令设计受到一定的限制。

2. Cache 结构与访问

UniCore 2 处理器的 Cache 采用四路组关联结构,分离的指令 Cache 和数据 Cache(以下均分别简称为 ICache 和 DCache)、ICache 和 DCache 均采用 Round-Robin 替换算法(也就是轮流替换一个组中的 4 个行,在面向移动终端应用类的处理器中,似乎很少见到该替换策略);提供同时或单独无效整个 ICache 和 DCache 的操作。DCache 的写策略可配置为写回(Write Back)或写穿(Write Through)。实行采用写分配(Write Allocate)的写失效策略;提供按物理地址索引的 Invalidate、Clean、Flush 单 DCache 行的操作。UniCore-2 采用的是比较传统的实 Cache 模式,Cache 的 Tag 和索引都采用实地址,也就是首先要通过 MMU 进行虚拟地址和物理地址的转换后再访问 Cache。理论上讲,这种方式性能略差而能耗也可能稍小。当前主流的处理器往往采用虚 Cache,CPU 发出的地址同时交给 MMU 和 Cache,虽然能耗有可能高一些,但性能也更好。

1) ICache 组织结构

ICache 为四路组相联映射,16KB 数据被划分为 4 个块,每个 ICache 体包含 128 行,每行存放 8 个物理地址连续的指令字,ICache 体内指令行的位置从 0 到 127 行递增,称为 ICache 行索引(ICache Line Index)。一条行索引可以同时选中 4 个 ICache 行,它们分别位于 4 个 ICache 体中,具有相同的索引(CPU 将根据访存地址中的若干位作为索引号来访问一个索引所对应的 4 个行,在 UniCore-2 中由于有 128 个索引行,因此需要地址中的 7 个比特作为索引)。

逻辑上,每个 ICache 行与一个 Valid 位和一个 TAG 字相对应;而实际上,ICache 中是用不同的存储体分别存放指令行和 TAG 信息。图 4-48 给出了不包含任何控制逻辑的和硬件通路的 ICache 体逻辑结构。

2) DCache 组织结构

DCache 为四路组相联映像,16KB 数据被划分为 4 个块,每个 DCache 体包含 128 行,每行存放 8 个物理地址连续的数据字,DCache 体内数据行的位置从 0 到 127 行递增,称为 DCache 行索引。一条行索引可以同时选中 4 个 DCache 行,它们分别位于 4 个 DCache 体中,具有相同的索引。

逻辑上,每个 DCache 行与一个 Valid 位、两个 Dirty 位(高半行和低半行各对应一个 Dirty 位)、一个 TAG 字相对应;而实际上,DCache 中是用不同的存储体分别存放数据行、TAG 和 Dirty 信息。图 4-49 给出了不包含任何控制逻辑的和硬件通路的 DCache 体逻辑结构。

当 Cache 被使能后,对所有指令或数据访存请求都会执行 Cache 查找(Cache Look-Up)。如果指令或数据已经在 Cache 中,则称为 Cache 命中(Cache Hit),其他情况称为 Cache 失效(Cache Miss)。SEP6200 处理器 Cache 能够全面地处理 ICache 和 DCache 在发生命中或缺失时的各种情况,完成基本的存储访问功能。

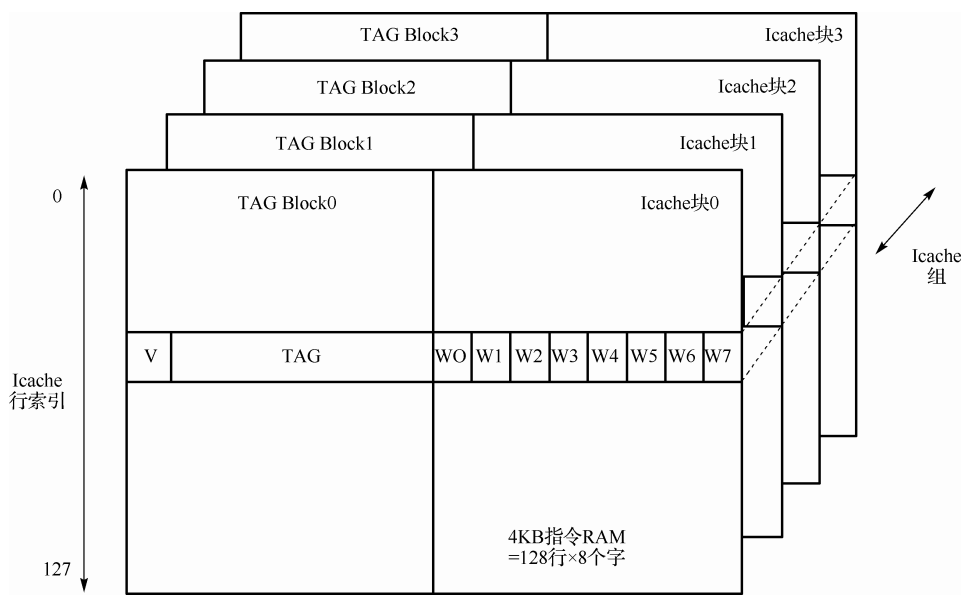


图 4-48 ICache 体逻辑组织结构

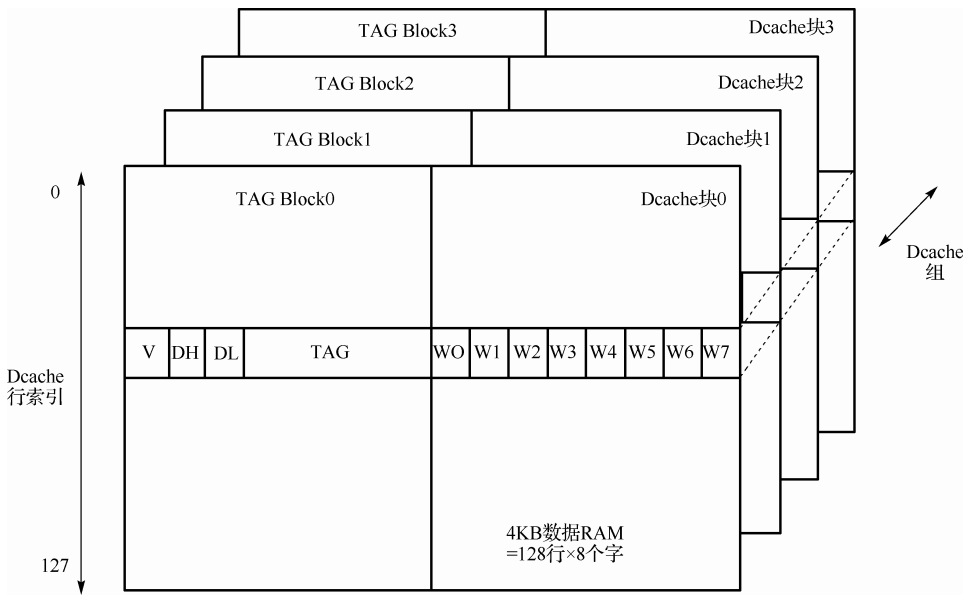


图 4-49 DCache 体逻辑组织结构

3. UniCore 架构指令集

UniCore 指令集是 RISC 类型的指令系统，包含如表 4-23 所示的指令类型。

表 4-23 UniCore 指令类型

数据处理类	数据传送指令
	算数逻辑运算指令
	比较指令
	定点乘法指令

续表

跳转执行类	无条件跳转指令
	条件跳转指令
内存相关类	装载指令
	存储指令
浮点指令类	浮点运算指令
	浮点比较指令

数据处理类包括数据传送指令、算术逻辑运算指令和比较指令。数据传送指令用于把立即数存入目的寄存器或者把寄存器值存入目的寄存器。算术逻辑运算指令实现了加、减、逻辑按位或、逻辑按位与和逻辑按位异或等运算，对应汇编代码为“OpCode rd, rs1, s2”，其中 OpCode 可以是 add、sub、and、or 和 xor；rd 代表目的寄存器；rs1 代表第一个源寄存器；s2 代表第二个源操作数，可以是寄存器、立即数、寄存器左移立即数、寄存器右移立即数等形式。比较指令不保存运算结果，只更新状态寄存器中相应的条件标志位。

跳转执行类包括无条件跳转指令和条件跳转指令，其可以分为相对 PC 值跳转与寄存器跳转。相对 PC 值跳转汇编指令为“b #imm24”，如果需要保存返回地址使用汇编指令“b.l #imm24”。寄存器跳转汇编指令为“jump rs”，如果需要保存返回地址使用“jump.l rs”。条件跳转汇编指令为“bcc #imm24”，需要判断状态寄存器中的条件位，如果条件成立就跳转，如果条件不成立就顺序执行。

与内存相关的指令是装载指令与存储指令，可以以 4 字节、2 字节或 1 字节为单位访问内存。不同单位有不同的地址对齐要求，以 4 字节为单位访问内存，要求地址的低 2 位为 0。由于复杂的寻址方式，使得装载指令与存储指令的寻址方式有多种形式。寻址方式由两部分组成，一部分为基址寄存器，一部分为地址偏移量。基址寄存器可以是通用寄存器，地址偏移量可以是立即数、寄存器、立即数及一个移位常数、寄存器及一个移位常数等形式。寻址方式的地址计算有 3 种：偏移量方法、事先更新方法、事后更新方法。偏移量方法中生成的内存地址为基址寄存器值和地址偏移量做加减运算的结果，但新的地址不写回基址寄存器（类似于 ARM 的前变址）。事先更新方法中生成的内存地址与偏移量方法相同，但在指令执行后，该内存地址将被写入基址寄存器（这类似于 ARM 指令的带回写的前变址）。事后更新方法中生成的内存地址为基址寄存器的值，在指令执行后，基址寄存器中的值与地址偏移量做加减运算，其计算结果存入基址寄存器（这类似于 ARM 指令的后变址）。

浮点指令类包括浮点运算指令与浮点比较指令，使用浮点数需要定义浮点数的格式，UniCore 浮点指令支持 IEEE754 标准的 32 位单精度和 64 位双精度格式。UniCore 架构有 32 个 32 位浮点寄存器，只允许使用 f0, f2, ..., f30 寄存器来保存单精度浮点数，相邻的两个浮点寄存器保存一个双精度数。UniCore 指令集除上述指令外还有一些杂项指令，如实现内存与寄存器直接交换数据的指令、条件执行加法的指令。

总的来说，UniCore-2 的指令集设计思想比较接近 ARM 的指令集，但在一些细节上还是有很多不同。比如，由于 UniCore-2 有 32 个通用寄存器，因此它的指令编码中，为了表达 3 个寄存器需要 15 比特，比 ARM 的 12 比特多了 3 位。这也许是因为 UniCore-2 的条件执行要比 ARM 指令集弱的原因。

4.4* 其他类型的计算引擎

4.4.1* GPU

1. GPU 的出现

我们不妨来回顾一下计算机图形界的发展历程——最开始的计算机屏幕只是一个圆滚滚的墨绿色小窗，上面移动的都是些白色的色块，它们构成了 PC 图形的最初搭配。后来，随着人们需求的不

断提升，屏幕里渐渐有了色彩，屏幕的分辨率也越来越高。渐渐地，色彩和分辨率的提升以及由此带来的对图形效果的不断追求终于累积到了 CPU 难以承受的地步。于是，GPU (Graphics Processing Unit) 从 CPU 中分裂出去了。

在移动计算领域，随着人们需求的不断变化，手机屏幕从计算器一般的液晶条慢慢变成了一个方形小窗，然后这个小窗有了色彩，彩屏的出现直接催生了更好的游戏和应用体验的出现，然后为了进一步提升体验，屏幕像素数也开始了攀升，这加大了手机 CPU 的处理压力。而屏幕更大（最新型号的智能手机的屏幕分辨率已经达到甚至超过 1920×1080）、应用环境更加开放的智能手机及平板电脑的性能需求终于让手机 CPU 走到了不堪重负的崩溃边缘。

PC 架构日新月异、眼花缭乱地发展着，架构性能是否高与重复单元的数量成正比，只要宏观结构不出现重大的失误，更多的执行单元和充足的资源以及控制电路一定意味着更加强劲的性能。在工艺确定的前提下，每个晶体管工作的环境，如亚阈值电压、载流子强度等都是有一定的，换句话说，性能与功耗之间基本上也是划等号的，更强劲的性能需要更多的运算单元、更多的缓冲以及更多的控制电路，也就需要更多的晶体管，自然也就会有更多的功耗需求。但是个人消费电子产品不同于 PC，它的便携移动性要求使得其整体体积不得不控制在一个非常小的范围之内，其电池容量不可能随心所欲地想做多大就做多，目前的锂离子电池已经达到了输出极限的瓶颈，电池成了手机及平板电脑处理器性能最沉重的枷锁。要解决功耗和性能的矛盾，同时满足高速增长的数据需求，途径只有一个——尽可能放弃通用性，将专门的任务交给专门的电路来完成，这个专门的处理单元就是 GPU。

2. GPU 的工作原理

早期的图形一般比较简单，如点、线、多边形等，这些图形是通过软件程序来处理和显示的。随着复杂图形的出现以及图形数据量的增大，通过软件来处理图形的速度已经无法满足人们的要求。由于图形处理是一个大数据量的过程，因此，可以设计专用的处理器来处理图形数据，即 GPU。GPU 的出现，使得计算机处理图形的能力得到很大提升。在图形处理方面，GPU 的能力远远超过 CPU。而 GPU 之所以能够很好地处理图形数据，其主要原因在于 GPU 的高度并行性和流水线设计。GPU 是一个多核处理器，其包含的内核数量成百上千，远远超过 CPU 的数量。

如图 4-50 所示的数据流图，3D 应用往往通过调用标准的 3D API 来绘制图形，比如微软的 Direct3D 以及开放标准的 OpenGL 等，不同的系统既可以通过软件来实现这些 API，也可以通过 GPU 来实现，甚至一部分 API 通过软件实现，另一部分通过 GPU 硬件实现。两者在速度上差别较大，对于同样的任务，GPU 处理数据的速度远远超过软件程序（在实际系统中，可能某些情况恰恰相反，比如通过 GPU 实现画点或画线的效率可能比用 CPU 还要慢。这可能是因为通过 GPU 驱动传递的数据开销比较大）。

1) 图形驱动

在 GPU 或者说 3D 硬件加速器发展的初期，由于缺少统一的标准支持，各个生产厂商的硬件设计区别很大，而用户想要使用某个产品也需要自己手动配置。生产商和用户都迫切需要一个统一的图形驱动来使得图形加速器更易于使用，兼容性更好。OpenGL 的出现弥补了这一空白。

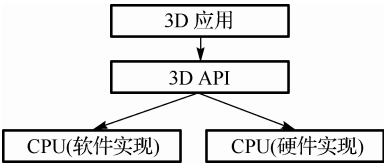


图 4-50 3D 数据处理流图

OpenGL 是 Open Graphics Library 的简称，是一个与硬件、与系统都无关的图形 API 集合。OpenGL 的应用十分广泛，不仅支持 2D 图形计算，也支持 3D 图形计算，既可以在 Windows 系统下使用，也可以在 Linux、UNIX 等系统下使用。另外，OpenGL 还推出了其针对移动多媒体领域的嵌入式版本——OpenGL ES。OpenGL ES 是针对嵌入式应用提出的，对功耗和性能方面做了权衡，支持多种手机操作系统，如 Android、iOS 等。

2) 图形处理流程

当 3D 应用程序将数据发送到 3D API 后，API 会将数据发送到对应的缓存，GPU 则通过指令缓存读取指令，并根据指令读取数据缓存的内容。从 GPU 的出现到现在，GPU 处理数据的流程也发生了变化，最初 GPU 采用固定管线(Fixed-function)的方式处理数据，然后出现分离的可编程(Programmable)管线的方式，而现在则基本采用统一的(Unified Shader)可编程架构的方式。

在固定管线的 GPU 中，所有的模块功能都是固定的，用户或者程序无法随意修改，数据都要存入指定存储位置，其处理数据的方式不可以随意变换，如图 4-51 所示。数据通过 API 函数发送到 GPU 的数据读取模块 Command Processor，然后该模块对读取的数据进行整理，分别发送到对应的模块，如图元的控制信号发送到 PA (Primitive Assemble)，而定点则发送到坐标变换模块 TL (Transform and Lighting)中。当定点完成坐标变换以及光照计算后，通过 PA 模块实现图元组装。组装好的图元经过 Rasterizer 后转化为 2D 平面上的点阵，也可以说是像素集合。产生的像素依次经过纹理贴图、颜色计算、雾计算、透明度计算、深度计算、混合以及抖动计算，最后输出到显示缓存 Frame Buffer 中。所有的模块功能都是固定的，而且处理顺序无法改变。

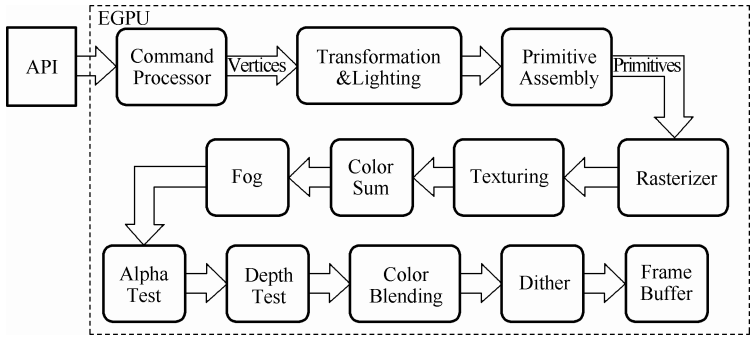


图 4-51 OpenGL ES 中的固定管线

简而言之，GPU 的图形（处理）流水线完成如下工作：

- (1) 顶点处理：这个阶段 GPU 读取描述 3D 图形外观的顶点数据并根据顶点数据确定 3D 图形的形状及位置关系，建立起 3D 图形的骨架。
- (2) 光栅化计算：显示器实际显示的图像是由像素组成的，我们需要将上面生成的图形上的点和线通过一定的算法转换到相应的像素点。把一个矢量图形转换为一系列像素点的过程就称为光栅化。例如，一条数学表示的斜线段最终被转化成梯状的连续像素点。
- (3) 纹理贴图：定点单元生成的多边形只构成了 3D 物体的轮廓，而纹理映射（texture mapping）工作完成对多边形表面的贴图。通俗地说，就是将多边形的表面贴上相应的图片，从而生成“真实”的图形。
- (4) 像素处理：这个阶段（在对每个像素进行光栅化处理期间）GPU 完成对像素的计算和处理，从而确定每个像素的最终属性。
- (5) 最终输出：一帧渲染完毕后，被送到显存帧缓冲区。

总之，GPU 的工作通俗地来说就是完成 3D 图形的生成，将图形映射到相应的像素点上，对每个像素进行计算确定最终颜色并完成输出。

而随着可编程着色器的提出，很多关键模块被设计为更灵活的编程模式。其中典型的可编程着色器包括顶点着色器(Vertex Shader)、片段着色器(Fragment Shader)，甚至有几何着色器(Geometry Shader)。着色器的提出使得 GPU 的灵活性大大提高了，关键模块处理数据的模式可通过编程改变，

不再局限于单一的数据处理方式，如图 4-52 所示。图中，Vertex Shader 可以完成坐标变换和光照计算，而 Fragment Shader 可完成纹理贴图、颜色计算、雾计算及透明度计算等。可编程模式下，GPU 处理数据的灵活性大大提高了，Shader 所完成的功能可以由用户通过编程进行改变。当然，其编程的范围是有限制的，比如 Vertex Shader 的编程只能针对 Vertex 数据，Fragment Shader 的编程也只能针对 Fragment 数据。

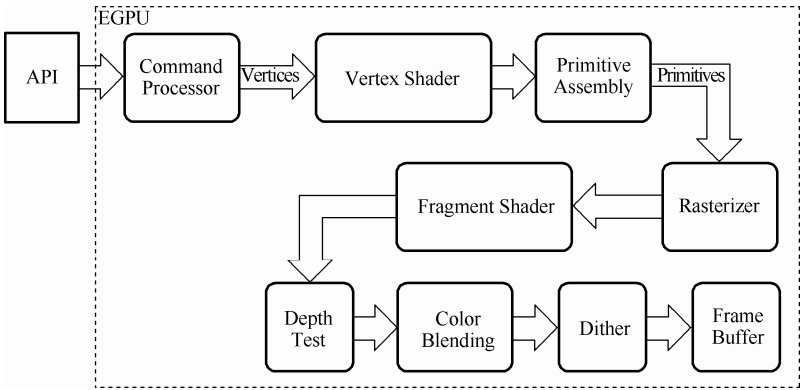


图 4-52 OpenGL ES 中的可编程管线

虽然可编程管线使得 GPU 的性能以及灵活性都得到了很大的提高，但是其资源的利用率在某些场景下会存在很大不足。在可编程的管线设计中，设计者需要预先估计不同种类的 Shader 的数量，例如设计中有多少个 Vertex Shader、多少个 Fragment Shader。而设计完成后，各种 Shader 的数量就固定下来了。由于 3D 场景应用十分广泛，不同的应用下其数据计算也会有很大不同，有的场景需要计算大量的顶点变换，而有的场景可能需要计算大量的像素点。因此，出现了 Unified Shader 的设计，即统一着色器。统一着色器的思想是，设计一种通用的可编程着色器，其可以完成各种通用计算，包括一些特殊的图形计算。这样的着色器既可以完成 Vertex Shader 的功能，也可以完成 Fragment Shader 的功能，区别仅在于运行的程序不同。这种统一着色器的设计方法很好地避免了由于场景多样性带来的资源利用不合理的问题。无论是定点处理复杂度高还是像素处理复杂度高的场合，都能将 Shader 合理地利用起来。采用统一着色器架构的 GPU 中，只有一种统一着色器，使得 GPU 的设计趋向于一种通用的处理器设计，只不过这种通用处理器包含很多图形学的专用算法指令。正是这样的设计，使得 GPU 在很多传统的计算领域取得了成功，从而出现了 GPGPU 的研究领域。由于现代 GPU 中集成了大量的并发计算单元，使得 GPU 本身的计算能力甚至超过了主 CPU，通用 GPU 计算（General Purpose GPU Computing）就是利用 GPU 的这些硬件资源实现图形运算以外的计算。由于 GPGPU 的兴起，使得异构计算（Heterogenous Computing）成为可能，不过目前学术界对此的研究重点还集中在传统的 PC 领域，关于嵌入式处理器的 GPGPU 研究才刚刚开始。

3. GPGPU 编程模型

GPU 的编程模型的发展和变迁与 GPU 硬件水平的发展紧密相关。早期的 GPU 流水线功能简单，只能执行比较固定的几类操作，并且都和图形处理密切相关，很难将非图形领域的应用映射到 GPU 上执行。即使在图形处理应用中，程序员也只能通过简单的图形 API（DirectX、OpenGL）进行程序映射，编程效率低下。随着 GPU 流水线功能的进一步完善和不断提高，出现了比图形 API 更高级的着色语言（HLSL、GLSL、Cg）。着色语言是在 C 语言的基础上扩充而来的，简化了编程的复杂性，提高了 GPU 的可编程性。但是，它们仍然没有有效屏蔽 GPU 的硬件细节，带有非常明显的图形处理痕迹，难以用于开发通用计算程序。

随着 GPGPU 的火热兴起，各主流 GPGPU 供应商也纷纷推出配套的编程环境，逐渐形成了几种主要的 GPGPU 编程模型，包括 AMD 的 Brook+、NVIDIA 的 CUDA 以及业界正在制定的覆盖 CPU、GPU 等多种计算平台的统一编程标准 OpenCL。下面对这几种业界提出的编程模型进行介绍。

1) Brook+

Brook+是 AMD 公司在 Brook for GPU 的基础上修订而来的，其最初的原型来自 Stanford 大学流处理器研究组提出的流编程语言 Brook。Brook 在底层的编程语言和运行库（如 OpenGL）的基础上进一步抽象，屏蔽了底层实现细节，提供了一种高级的编程接口。它基于 C 语言进行扩展，引入了流（Stream）和核心（Kernel）两个重要的概念。流是指一组结构相同的可以被并行操作的数据对象，核心函数则是指由程序员自定义的施加在流元素上的操作。核心函数隐式地并行施加于所有的流元素之上。Brook for GPU 是 Brook 的 GPU 版本，引入了一些针对 GPU 硬件的特性描述，其编译器和运行环境将 GPU 抽象为一个流处理器，从而将流编程模型引入 GPU，有效地支持了 GPU 的通用计算。AMD 在 Brook for GPU 的基础上针对其高性能 GPU 做了很多修订，推出了 Brook+编程模型。

图 4-53 所示为 Brook+软件结构框图。Brook+源码经过分离，产生 CPU 部分代码和 GPU 部分代码，其中 CPU 代码交由本地 C++编译器编译执行，而 GPU 部分则交由 Kernel 编译器编译产生在 GPU 上运行的可执行代码。CPU 代码以函数调用的形式调用核心函数完成计算任务。

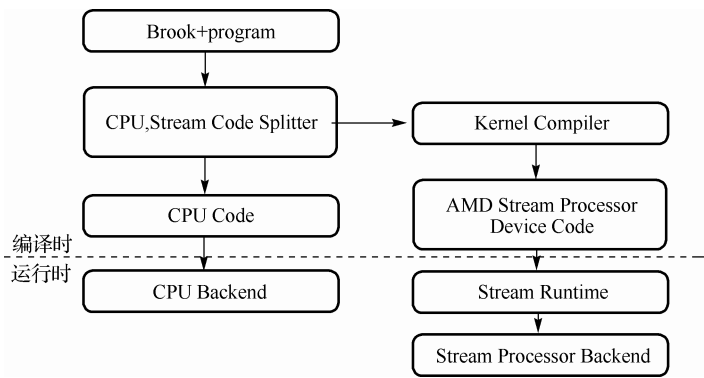


图 4-53 Brook+编程环境框架

在 GPU 内，核心函数以 SIMD 方式并行执行于所有的流元素之上，执行的范围称为执行区间（Domain），GPU 为执行区间内每个点都派生出一个线程，并调度到某一个线程处理器上执行，如图 4-54 所示。线程执行的内容就是核心函数。程序员的主要工作就是构造并行执行的计算区间，以及区间内每个点所执行的操作，而诸如线程派生、线程分配和调度、存储访问等操作都是由 GPU 硬件自动完成的，从而屏蔽了 GPU 硬件细节，极大地简化了 GPU 通用计算的开发。

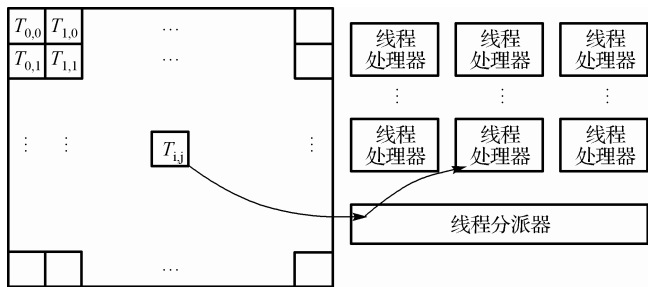


图 4-54 Brook+并行计算模型

2) CUDA

CUDA 全称为统一计算设备框架（Compute Unified Device Architecture），是 NVIDIA 公司针对其 GPU 设计提出的 GPGPU 编程模型，它也以 C 语言为基础，降低了编程难度。在 CUDA 计算模型中，应用程序分为 host 端和 device 端，前者是指运行于 CPU 上的部分，后者则是运行于 GPU 上的部分，和 Brook+ 一样，这部分代码称为核心（Kernel）函数。CPU 代码准备好数据后，将数据存放到显卡的存储器中，再调用核心函数进行执行，执行完毕后再由 CPU 代码将计算结果复制回主存。

在 CUDA 计算模型下，GPU 执行的最小单位是线程，多个线程组成一个线程块，线程块中的线程可以共享一片存储器，并以任意顺序执行，在硬件资源受限的情况下甚至可以串行执行。一个核心程序由一个或多个线程块组成，一个应用则由若干个核心程序构成。图 4-55 给出了各计算实体的对应关系。

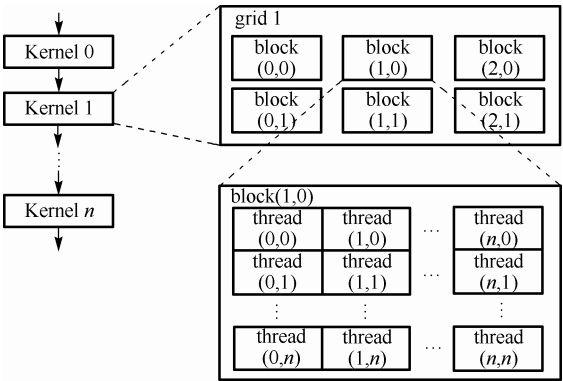


图 4-55 CUDA 计算模型示意图

3) OpenCL

OpenCL 是一个业界正在制定的统一编程标准，它可以覆盖通用 CPU、GPU、DSP 芯片、手持设备等多种计算设备，将这些异构的计算资源纳入一个统一的框架中，向程序员提供一组 API 和运行库，使得程序员可以透明地使用这些设备。

和 CUDA 类似，OpenCL 的计算平台由一个主控设备（Host Device）和一组计算设备（Compute Device）组成，OpenCL 主控程序运行于主控设备之上，并向各计算设备提交计算命令；计算设备上运行核心（Kernel）程序，完成计算后将结果返回给 OpenCL 主控程序。通常情况下，OpenCL 的计算设备都由一个或多个计算单元（Compute Unit）组成，而每个计算单元又包含多个处理单元（Processing Unit），它们组成了一个三层的嵌套关系，提供了大量潜在的并行性。主控设备在向计算设备提交一次核心计算时会指定一个执行的区间，计算设备会为这个区间内所有的点都派生出一个线程来执行。每一个线程称为一个工作单元（Work-item），而多个工作单元可以被分组，称为一个工作组（Work-group）。一个工作组上的所有工作单元可以共享一块局部存储器。

计算设备上运行的核心程序可以访问 4 类存储对象，包括全局存储器、常数存储器、局部存储器和私有存储器。全局存储器对于所有工作组内的所有工作单元都可以进行读写访问，并且可以访问存储对象的任意位置。常数存储器存放核心程序执行所需要的常数参数，由主控程序负责分配和初始化。局部存储器则隶属于一个工作组，只对组内和工作单元可见。私有存储器是属于一个工作单元的存储单元，不对其他工作单元可见。各类存储对象在计算设备上的分布和使用情况如图 4-56 所示。

4) 编程模型对比分析

以上介绍的 3 种 GPGPU 编程模型中，Brook+和 CUDA 属于设备相关的编程模型，仅支持相关厂商推出的 GPGPU，而 OpenCL 则属于设备无关的编程模型，对各种计算设备进行了一种统一的抽象。

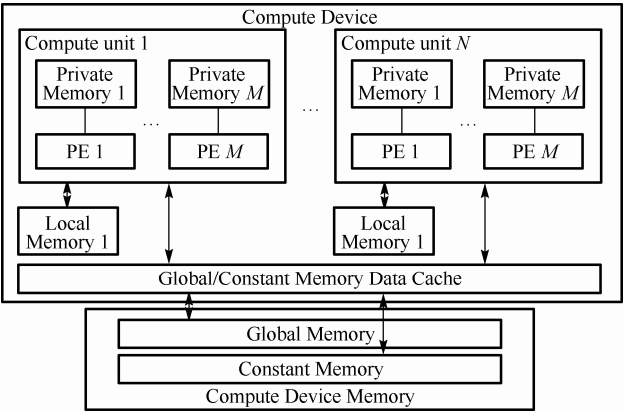


图 4-56 OpenCL 存储模型

从计算与数据的关系来说，Brook+属于一种典型的流编程模型，相对 CUDA 和 OpenCL 具有更明显的以数据为中心的计算模型。Brook+程序中的数据被定义为流，计算空间是按照输出流的空间派生出来的，计算空间和流空间一一对应，一个计算点一般只计算流中的一个数据。对数据流的随机访问需要将流设置成专门的访问类型。与之相对，CUDA 和 OpenCL 则不存在这类限制，它们直接将计算空间的组织交给程序员完成，更加贴近于传统的多线程编程模型，由程序员指定线程空间以及每个线程完成的操作，而数据默认都可以通过下标进行随机访问。

4.4.2* 可重构计算

1. 可重构计算的出现

在嵌入式系统快速发展的今天，以交互式媒体处理、高性能移动计算等为代表的新一代应用，对嵌入式终端的性能、功耗、灵活性、集成度和成本等同时提出了苛刻的要求。从传统的以通用处理器（GPP）为基础的软件执行方式，到以专用集成电路（ASIC）为代表的硬件执行方式，已很难满足上述要求。GPP 虽然具有很强的灵活性，但指令流驱动的执行方式、有限的运算单元和存储器带宽使其整体性能和功耗并不理想；ASIC 虽然具有很高的能量效率，但低下的灵活性使其很难满足层出不穷的应用需求，高昂的 NRE 费用^①更使其无法在纳米工艺时代（例如 65nm 以下）得到大面积的推广。可重构计算（Reconfigurable Computing）正是在这种背景下出现的一种将软件的灵活性和硬件的高效性结合在一起的计算体系结构，其在性能、功耗和灵活性等关键指标之间具有更好的平衡，填补了通用计算和专用计算之间的空白，正日益获得越来越广泛的研究和应用。

在目前流行的图形、图像和视频编解码算法中，存在诸多适合并行计算的共性特点，可以归纳为基于块和宏块的并发操作、规则的数据依赖关系及计算复杂度高度集中这三个方面。

1) 基于块和宏块的并发操作

在运动图像系列标准中，基本的编解码操作都是基于宏块（16×16 像素）和块（8×8 像素）来进行的。在 MPEG2 和 MPEG4 标准中，基于宏块进行运动估计和运动补偿来压缩时间冗余，基于块进行 DCT 和 IDCT 来压缩空间冗余。AVS 标准在此基础上引入了 16×8、8×16 的块操作模式，而 MPEG4 AVC/H.264 标准中还引入了 8×4、4×8、4×4 的编码模式。在 JPEG 标准中，采用基于 8×8 像素块进行 DCT 和 IDCT 来编解码，JPEG2000 标准最重要的特点则是将 JPEG 标准中的 DCT 换成了 DWT（离散

① 也就是为了研发、生产这款 ASIC 芯片所需要投入的一次性费用，其中研发成本和量产时芯片的掩模（光罩）成本是 NRE 的主要组成部分。

小波变换)。通过对运动图像系列标准和静止图像系列标准进行比较可以发现, JPEG 中的编解码技术和 MPEG2 中的 I 帧编解码技术基本一致。如果将 MPEG2 中的图像组全部都定义成 I 帧的话, 其编解码方法基本与 M-JPEG 一致。总的来说, MPEG 和 ITU 的系列视频解码标准在基本原理、核心的编码技术和算法的体系架构上十分类似, 只是在编解码的复杂程度、算法中的具体参数以及个别算法等方面存在差别。而 JPEG 系列标准可以认为是 MPEG 系列标准的特例。这些标准之间相互包含, 各有特点和差异。但是, 这些差异并不影响上述标准和算法在并发执行时具有的共性——都是基于块和宏块进行操作。由此可以得出, 在图形、图像和视频编解码应用中, 可以实现比指令级并行更高的并行性操作——基于块和宏块的并发执行操作。

2) 计算复杂度的高度集中

在 MPEG 系列标准中, 虽然标准中包含了多种算法, 但是主要的计算量集中在少数算法中。例如, 在 MPEG4 编码 (QCIF 格式, 采用 DSP 处理器实现) 器中, 运动估计占了 66% 的计算量, 运动补偿占了 15% 的计算量, DCT 和 CAE 占了 13% 的计算量。在 MPEG4 解码器中, 运动补偿占了 10% 的计算量, IDCT 占了 9% 的计算量, MB-padding 占了 14% 的计算量, Deblocking 和 IQ 占了 13% 的计算量, RVLD 和访存占了 40% 的计算量。虽然上述统计针对不同标准会有所差别, 但是可以得出的结论是, 在图形、图像和视频编解码应用中, 只需针对几个重点算法进行算法的并行性挖掘, 即可提高整个标准的并发执行效率。

3) 规则的数据依赖关系

在图形、图像和视频编解码中, 一个显著的特点是编解码算法所需的数据流十分规整, 数据依赖关系简单明确。由于图像的编解码都是基于宏块和块进行操作的, 因此一个块内所有的数据都有相同的数据依赖关系, 在编解码时可以简单地进行块级的数据操作, 无需针对某个像素或某几个像素频繁和难以预测地进行复杂的访存操作。由此可知, 在图形、图像和视频编解码应用中, 可容易地进行块级的数据流组织和搬移, 其访存结构简单, 访存开销较低。

基于图形、图像和视频编解码算法, 人们研究了各种适合上述特点的体系架构, 其中最为看好的即是基于可重构阵列的媒体处理器架构。基于可重构阵列的媒体处理器可以很容易地解决阵列硬件规模与算法计算规模的匹配问题, 可以较灵活地实现块级和宏块级并发执行操作。同时, 基于可重构阵列的媒体处理器简化了数据流的组织难度, 其访存代价比基于指令的媒体处理器小得多, 而通用性相当, 优势明显。

当前的图形、图像和视频编解码算法的研究中, 主要集中在 DVC (Distributed Video coding) 技术、3-D 视频编解码技术、超低功耗编解码技术、超大图像 (如 SHTV 超清视频) 编解码技术等研究领域。总的来说, 上述研究并没有颠覆传统的视频处理技术架构, 其图形图像处理算法的原理、方法, 其算法内在的并发性和规则的数据依赖关系等特点没有改变。由此可以预测, 基于可重构阵列的媒体处理器技术不仅是解决当前图形、图像和视频编解码计算的关键技术, 也将在未来的图形、图像和视频编解码计算中起着举足轻重的作用。

可重构计算的概念产生自 20 世纪 60 年代, 加州大学洛杉矶分校的 G.Estrin 教授在其里程碑式的文章中首先谈到这一概念, 即计算机可以通过一个主处理器加上一组可重构硬件来组成。主处理器负责控制可重构硬件的行为, 可重构硬件可以剪裁、重组, 执行某一特定的任务。在可重构硬件上执行的任务结束以后, 这部分硬件又被重新配置, 从而可以执行其他任务。由于这个先进的概念远远超前于当时的技术水平, 直到 20 世纪 90 年代之后, 这一研究领域才重新获得人们的重视, 各种不同的可重构计算技术层出不穷。人们从不同的研究和应用角度出发, 对可重构计算体系结构的理解也不尽相同。比较公认的定义是由加州大学伯克利分校的可重构技术研究中心于 1999 年提出的一种广义的定义, 将可重构计算体系结构视为一类计算机组织结构, 具有区别于其他组织结构的两类突出特点:

①制造后芯片的可重构能力（区别于 ASIC）；②能实现很大程度的算法到计算引擎的空间映射（区别于通用处理器）。凡具备以上两个特点的计算方式都属于可重构计算的范畴。

可重构媒体处理软硬件架构正是符合了上面的特点。除此以外，可重构媒体处理软硬件架构的特征还包括：①算法的控制流与数据流分离：数据流由可重构计算引擎处理，处理器执行控制流并负责可重构计算引擎的重构；②可重构计算引擎多采用 PE（Processing Engine）阵列的结构实现。这种结构具有动态配置的特性，并且能实现算法、指令、数据多个层次上的并行度。

2. 可重构计算的国内外研究现状

近年来，国际上面向媒体处理的可重构处理器主要有 PACT 公司的 XPP-III、IPFlex 公司的 DAPDNA-2 和 IMEC 的 ADRES 处理器等。XPP-III 是 PACT 公司 2007 年推出的一款用于信号处理的软件可编程独立处理器。它把粗颗粒度 PE 阵列与用于控制流代码的连续架构结合起来，将传统架构中的处理器部分弱化了，重点着眼于数据流的连续处理，可以将复杂的算法（如 H.264）进行整体的映射。XPP-III 内核工作在 350MHz 的时钟频率下，能以 1920×1088 的分辨率执行 H.264 Main Profile 的处理，用 90nm 工艺实现，最高可以工作在 450MHz，面积为 75mm^2 ，解码 HD（ 1920×1080 ）达到了 24fps。DNP DNA-2 是日本 IPFlex 公司和富士通公司于 2004 年联合推出的动态可配置处理器。它采用传统的 RISC 核加上 PE Matrix 的架构，具备动态可重构的特点。DAPDNA 有 4 个配置信息槽（一个当前的，三个后台的），配置信息也可以从片外读取，后台和当前配置之间的切换可以在一个时钟周期内完成。多套数据接口和特定的片上网络设计使得 PE 阵列的数据吞吐率在 16Gbps 以上。IMEC 在 2005 年采用 90nm 工艺实现的 ADRES 处理器也是媒体处理 SoC 平台的很好选择。IMEC 开发的媒体芯片（芯片面积 75mm^2 ）中集成了多个 ADRES 核（ 50mm^2 的 ADRES 核），功耗为 700nW，支持 HDTV-level 的 AVC 和 VGA-level 的 SVC。单个 8×8 的 32 位 ADRES 核，在 90nm 工艺下测试 H.264/AVC decoder，能耗效率为 50~60MOPS/mW，峰值性能 25GOPS。C 代码在 ADRES 上的执行效率用周期数统计比当时最新的 DSP 高 10 倍以上。此外，可重构处理器也被广泛应用于通信领域。Freescale 于 2004 年推出的 MRC6011 处理器是一款商用可重构阵列处理器。MRC6011 由控制器和 6 个 PE 阵列核构成，拥有两套独立的存储器接口及两套输入和布线的接口。 $0.13\mu\text{m}$ 工艺下可工作在 250MHz，功耗 3W。Intel 于 2004 年在其设计的重构通信阵列（一个采用路由连接的 4×5 的异构 PE 阵列）上实现了 802.11a、和 3G、4G 的标准。2007 年的 ISSCC 会议上，Intel 发表了其最新设计的可重构维特比译码加速器。

可重构媒体处理软硬件架构在国内的研究也比较活跃。西北工大把可重构阵列作为一个协处理器进行研究，实现了部分图像处理算法（如 DFT、离散二维卷积运算等）和视频压缩算法（DCT、ME 等）在 PE 阵列上的并行化。中科大研究了可重构计算体系结构，提出了基于“ARM+FPGA”的软硬件架构，并在 Xilinx VirtexII 开发板上进行了实验测试。东南大学和清华大学合作设计的 REMUS-II 可重构计算体系结构，由 RPU（用于完成媒体处理任务中的计算密集型部分）+ μ PU（生成配置包以及进行不适合 RPU 完成的控制密集型操作）组成，可实现 H.264、AVS、GPS 等多种媒体与通信算法的映射。在可重构处理器体系结构上，国内和国外的方案并没有本质的区别：采用 RISC 处理器或者其他控制模块和配置模块加上 PE 阵列，组成可重构处理器的基本结构。

国内在可重构 PE 和 PE 阵列上也有一定的研究。合肥工大提出了一种可重构协处理器 DreAC 架构，集成了 8×8 的通用 ALU 阵列、数据缓冲区和中央控制单元。南京大学也提出了一种粗颗粒度的动态可重构计算系统 MPRS。国防科大则给出了面向循环程序的固定指令多数据流计算模型 FIMD，适用于 PE 阵列中的高性能计算单元。深圳大学提出适于集成化的 CORDIC-CELL 结构，十分适合构建单芯片的运算阵列。东南大学和清华大学合作开发的 REMUS-II 可重构计算架构对 RPU 的配置信息采用了三层分层机制，以减轻配置信息存储及传输的压力。

通过分析国际上对可重构处理器体系架构,我们发现“控制单元 + 可重构 PE 阵列”的架构将长期沿用,高速的动态配置和运算的并行化将成为对可重构处理软硬件架构的新兴研究方向,并且会不断深入。软硬件架构的发展方向之一:是在 RISC 和 PE 阵列之间插入管理、编译的硬件模块,将应用程序的划分映射算法在片上采用硬件的方式来实现,进一步提高灵活性。典型的代表有加州大学 Riverside 研究的 Warp 处理器结构;软硬件架构方向之二:是集成专门的硬件控制器来控制 PE 阵列的运行,不断弱化微处理器的作用,以有效控制功耗,例如 PACT 的 XPP-III 和 Intel 的重构通信阵列;软硬件架构方向之三:既然可重构阵列可以实现各种算法并且被动态配置,那么阵列中可能存在着冗余。该冗余的阵列可以用来提供安全性的校验电路,也可以用来执行代码的编译和 PE 阵列的映射操作。这使得芯片的计算方式与以往有很大的不同,可重构阵列可以实现对自身的配置和调度,算法的编译可以被硬件自动实现,这有可能严重挑战传统的硬件电路计算方式。

3. 可重构计算架构特征

1) 主控核特征

在典型的可重构处理器中,通常在可重构阵列的基础上耦合一个传统的微处理器作为主控核,其作用主要包括以下方面:

首先,由于可重构阵列只适合于运行结构规则、数据依赖性弱的数据密集型任务,而对于变长循环、分支控制等控制流任务难以实现并行计算,因此一般将此类任务映射至主控核中完成。

其次,可重构阵列需要针对不同任务完成可重构单元和互联组织的重新配置,这些配置工作一般也由主控核完成,可通过总线以 DMA 方式实现配置信息的传输或者以指令方式进行配置信息的加载。具体主控核参与重构配置的方式和主控核与可重构阵列的耦合方式以及配置控制器等因素有关。

典型的可重构处理器采用的主控核按照其架构类型可分为 RISC、VLIW 和 SuperScalar 3 种。由 RISC 类主控核构成的可重构芯片主要面向计算的通用性要求较高的领域,比如同时需要对数据密集型、计算密集型以及高数据吞吐量的运算做加速的领域。而由 VLIW 类主控核构成的可重构芯片主要面向各类 DSP 变换,一般是用于对传统 DSP 的改造(现在看来多用于无线通信领域),增强其算法映射的灵活性。使用 SuperScalar 类主控核的可重构处理器主要是考虑增强多线程的并发性。

下面对这 3 类主控核分别做较为详细的分析与比较。

(1) RISC。

- ARM: 指令集较为复杂,但通用性强。目前使用 ARM 作为主控的粗粒度可重构处理器比较少(欧洲现在的 MORPHEUS 项目是 ARM926+XPP 架构,创新公司 ZII 芯片是双核 ARM926EJ+48 PEs)。事实上,ARM 留有较好的指令扩展机制,也可以通过协处理器的方式支持新的指令,而且 ARM 有较为完备的编译环境(ARMCC 和 GCC for ARM),比较容易进行工具链的探索。从建模的角度讲,ARM 有大量开源的内核模型,包括 SimpleScalar、Skyeye、Facsim 等,非常容易实现快速原型。
- MIPS: 指令集简单,方便裁剪,也容易扩展指令集。已知的粗粒度可重构处理器的 RISC 类主控核大多选用 MIPS(但大多对其做了一定的裁剪与修改)。比如,MorphoSys(使用 TinyRISC,其实质是 MIPS,增加了一组用于配置重构阵列及互联网络的指令)、MATRIX、GARP、REMARK、RAW 等。

(2) VLIW。

编译器难度大,性能高,灵活性差,目前已知的可重构处理器也仅有 ADRES 和 XiRISC 选用 VLIW 作为其主控核,其中 ADRES 的主控核是一款多核可重构的 VLIW 处理器,利用多核的特性可以实现进程级的并行性,利用重构技术可以增强普通 VLIW 的灵活度,同时由于其内核和重构阵列共用一组

寄存器，实现了数据在主控核和重构阵列间的高速通信，是一款真正意义上紧耦合的粗粒度可重构处理器。而 XiRISC 处理器是对传统 DSP 芯片的扩展。

(3) Superscalar。

硬件设计难度大，灵活度大，目前有几款可重构处理器采用超标量实现指令级并行，比如 Spyder 等。事实上，大多数通用的 RISC 处理器都支持 Superscalar 方式，比如最新的 ARM 以及 MIPS 处理器。利用超标量技术可以实现多条指令的同时发射，并借此加速通用程序中控制流的处理（关于超标量技术，可以参阅 4.1.5 节）。

2) 耦合方式特征

主控核与可重构阵列间的耦合方式影响着可重构处理器的工作性能与配置效率。从耦合度的角度进行划分，典型可重构处理器的耦合方式分为 5 种类型，分别为片外器件方式耦合、总线方式耦合、协处理器方式耦合、功能模块方式耦合和内嵌方式耦合。

图 4-57 所示的可重构阵列作为一个片外独立器件与主控核耦合。在这种耦合方式中，主控核与可重构阵列通过主控核的 I/O 接口完成数据与配置信息的传输，因此传输速度非常慢，只适合于可重构阵列完成大量数据处理的情况，主要应用于仿真系统中。

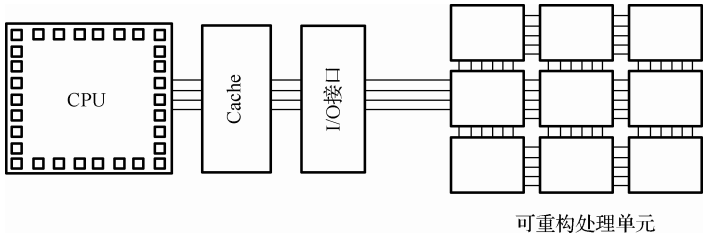


图 4-57 片外器件方式耦合

图 4-58 所示的可重构阵列作为一个总线设备与主控核耦合。在这种耦合方式中，可重构阵列的工作模式类似于一个硬件加速器，通过总线协议与主控核完成输入 / 输出数据与配置信息的传输。在具体实现中，通常采用此类耦合方式的可重构阵列替代原有总线中的硬件加速器。

采用此类耦合方式的典型可重构处理器包括 MorphySys、XPP 等。

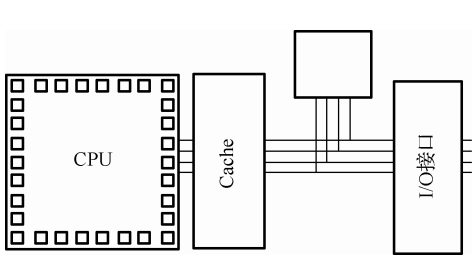


图 4-58 总线设备方式耦合

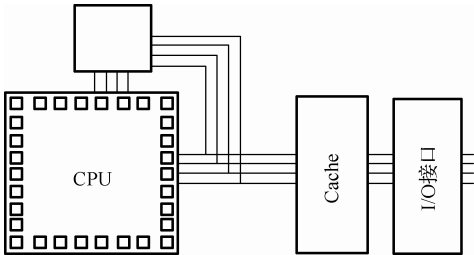


图 4-59 协处理器方式耦合

图 4-59 所示的可重构阵列作为一个协处理器与主控核耦合。在这种耦合方式中，需要对主控核扩展指令集，以完成对可重构阵列的控制。这种耦合方式减小了主控核与可重构阵列之间的通信开销，可重构阵列通过接收指令独立完成相关运算，在此过程中无需主控核的干涉。

采用此类耦合方式的典型可重构处理器包括 ADRES、REMARC、Zippy 等。

图 4-60 所示的可重构阵列作为一个独立的功能模块耦合于主控核内部。在这种耦合方式中，可重构阵列成为主控核中数据通路的一部分，完成部分特定逻辑的运算。

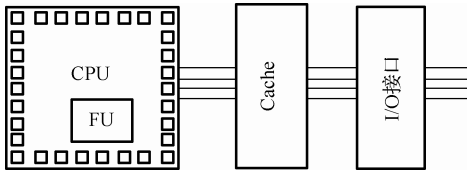


图 4-60 功能模块方式耦合

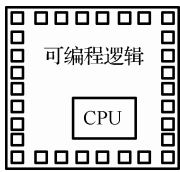


图 4-61 内嵌方式耦合

在图 4-61 所示的可重构处理器中，主控核作为可重构阵列的一部分，以软核或硬核的形式内嵌于可重构阵列中。

此类耦合方式主要应用于 FPGA 等细颗粒可重构阵列中。如 Altera 公司采用 NIOS 和 NIOS II 作为 Stratix 系列 FPGA 的软核，Xilinx 公司采用 PowerPC 405 处理器作为 FPGA 的硬核。

主控核与可重构阵列之间的耦合程度影响着可重构处理器的性能。耦合程度越高，主控核与可重构阵列之间的通信开销越小，主控核可以更频繁地通过可重构阵列完成数据密集型任务的处理，但同时也导致可重构阵列需要主控核较多的干涉。耦合程度越低，主控核与可重构阵列之间的并行程度越高，但主控核与可重构阵列之间的通信开销随之增加。

3) 可重构阵列特征

(1) 可重构单元特征。

可重构单元（被称为 PE、RC、rDPA、basic FU、RPA unit 等）是粗粒度可重构处理器的核心，是数据通路上负责实现具体操作、运算以及控制的部件。

① 可重构单元颗粒度。

颗粒度是可重构处理器的一个核心参数，传统的细粒度可重构处理器由于其只能单比特地处理数据和配置片上通信资源，因此虽然能实现任意逻辑（也就是适用于对通用计算的加速），但细粒度可重构处理器受制于以下几方面的缺陷，使其难以满足手持嵌入式计算设备的要求：

- 细粒度可重构将大量芯片面积消耗在互联路由上，这将导致较低的硅片利用率。不仅如此，使用基于开关网络的路由（switched routing wries）相较于硬连接（hardwired connections）更消耗系统能耗。
- 细粒度可重构需要大量的配置存储器以存储大量 PE 和互联的配置信息，同时重构时间开销以及能耗开销也都将增大，这可能带来另外一个问题，即重构是否能在确定时间内完成，即重构实时性的问题。
- 细粒度可重构系统基于 FPGA 的实现方式非常类似于传统 VLSI，因此需要硬件编程专家利用硬件描述语言实现对 PE 和路由的映射，而传统的软件高层语言（比如 C 语言）很难直接映射到 FPGA 上。

与之相对，可以处理大于 1 比特的复杂操作的粗粒度可重构处理器可以在更好地满足系统对面积、能耗以及重构开销的基础上，针对流类型应用加速，也就是更适用于专用处理。

对于粗粒度可重构处理器而言，其粒度大小是数据通路的带宽，包括 PE 的带宽和片上网络的数据带宽。不能简单地认为片上互联结构的数据带宽就一定都与 PE 的带宽一致，事实是所谓可变粒度就是指 PE（主要由 ALU 构成）可以组合起来，处理更大带宽要求的情况。

一般粗粒度可重构处理器的带宽为 4 比特、8 比特、16 比特、32 比特、64 比特或者 128 比特，具体的选择和实际需要加速的上层应用关系较大，因此将粒度作为参数，将能做大量设计空间探索方面的研究。事实上，粒度的确定可以看作是重构系统效率和灵活性的折中（tradeoff）。较大的数据宽度可以方便地在 PE 上映射整字的操作，但某些应用要求对部分数据做低于单个字长的操作，此时就需要引入额外的拆分过程，导致效率降低。相反，较小的数据宽度可能不存在拆分过程，但其实现较宽数据的操作时必须同时使用多个 PE，这将增加路由的成本。

表 4-24 罗列了不同现存粗粒度可重构处理器的粒度。

表 4-24 粗粒度可重构处理器的粒度

发布时间	系统名称	阵列布局	重构模型	数据宽度
1990	PADDI	Crossbar	Static	16 比特
1993	PADDI-2	Crossbar	Static	16 比特
1994	DP-FPGA	Mesh	Static	4 比特
1995	KressArray	Mesh	Static	32 比特
1996	Colt	Mesh	动态	16 比特
1996	RaPiD	线性阵列	大部分静态	16 比特
1996	MATRIX	Mesh	动态	16 比特
1997	Garp	Mesh	Static	2 比特
1997	Raw	Mesh	Static	32 比特
1998	PipeRench	线性阵列	动态	4 比特
1998	REMARC	Mesh	Static	16 比特
1998	MorphoSys	Mesh	动态	16 比特
1999	CHESS	Mesh	Static	4 比特

② 可重构单元支持运算。

可重构单元一般可支持以下运算。

- 基本运算。包括基本的算术运算、逻辑运算、求绝对值运算、求最大/小值运算、移位运算、乘加运算等。
- 特殊运算。包括预测运算、赋值（路由）运算等。

③ 可重构单元构成。

- ALU：浮点或定点 ALU，可以实现如下功能：逻辑运算、算术运算、求绝对值运算等 25 种简单运算以及单周期的乘加运算。
- 乘法器：完成乘法运算。
- 移位器：可以用于结果的精度控制等。
- 多路选通器：完成多输入的选择，即其他的 PE、局部存储器、内部寄存器以及常量等。
- 内部寄存器：用于保存配置信息和部分中间数据。
- 上下文寄存器：定义 PE 功能以及路由。
- MAC 模块：主要面向 DSP 应用中大量的乘加运算，可执行浮点的复数运算。
- 细粒度单元：实现定制的功能单元 FU。
- 局部存储器：用于中间数据的保存，可以实现为 FIFO、Register、RAM 等。

④ 可重构单元强度。

可重构单元的强度（strength）表现为可重构单元中各功能单元的个数。由于互联组织存在延时，因此和采用多个可重构单元实现功能单元相比，在同一个可重构单元中实现多个功能单元在性能上更具优势。

Bansal 等人在文献中对两种不同结构的可重构单元阵列进行比较。第一种为 8×8 的可重构单元阵列，每个可重构单元包括 1 个 ALU；第二种为 4×4 的可重构单元阵列，每个可重构单元包括 4 个 ALU。虽然这两种可重构单元都包含 64 个 ALU，每个 ALU 支持的操作类型相同，但实验结果表明第二种可重构单元阵列具有更好的性能，其原因即为可重构单元中各功能单元之间的通信延时比可重构阵列中各可重构单元间的通信延时要短。

（2）数据存储器特征。

本节讨论的存储器指的是可重构阵列中缓存中间结果数据的存储器。在可重构阵列中，为了减少阵列外数据的访问量，减少数据访问时间，降低数据传输功耗，通常尽量将中间结果数据存储在可重构阵列中。

可重构阵列中的存储器按照其实现方式可以分为 3 类：

- 在可重构阵列外，和重构阵列紧耦合（比如 MorphoSys 的 Frame Buffer）。
- 在可重构阵列中作为一种特殊的可重构单元存在（比如 XPP3 的 RAM PAE）。
- 在重构单元内部（具体实现形式可以是 FIFO、Register、LUT 或者 RAM 等）。

（3）互联组织特征。

互联结构是决定可重构处理器灵活度、成本以及性能的关键因素之一，按照通用的分类方式，我们可以将其大致分为两类：

- 动态互联。常用于并行计算机之间或者芯片内部多处理器之间的连接（NoC）。动态互联通过路由器连接处理单元或者部件的数据输入/输出通道，通道上的数据含有路由信息，路由器根据在数据链路上的报文目的地，通过路由和报文仲裁，动态选择报文的路由通道传到下一个路由。虽然没有物理信号上的直接连接，但通过路由器间接连接，构成了一种逻辑上的连接关系。RAW 正是采用这类互联方式的。
- 静态互联。静态互联是一种物理连接，只要配置好后 PE 之间的连接关系和通信关系便不会改变（类似于 FPGA 中的硬连接），逻辑块之间的连接通过开关矩阵完成，实现时由传输晶体管在两条连线的交叉点上控制两条连线的通断。

主流的可重构处理器的重构阵列的互联方式都为静态互联，因此我们主要考查此类互联类型。静态连接按照连接的具体类型又可以分为如下 3 类：

- 点对点互联。所谓点对点互联，是指重构单元直接存在两两间的直接相连，按照互联的可能性可以分为 4NN 和 8NN 两类，前者为中心重构单元与上下左右 4 个重构单元形成直接互联关系，而后者则需要考虑左上、右上、左下以及右下等额外 4 类位置关系。
- CrossBar 型。可以实现基本的连接仲裁，比如 PADDI 类型的重构处理器。
- 总线互联。包括纵横连线和全局连线。这里的总线是一种局部总线，仅负责形成数据通路。按照连接的距离还可以分为分段式的总线互联与整段互联。

互联结构一般包括 3 大类：NN 型、SwitchBox 型和总线型。其中，NN 型按照其连接的 PEs 又可分为 4-NN 型、8NN 型，按照其连接的距离又可分为点对点、跨 1 个 PE、跨 N 个 PE 等。而总线型一般又可分为局部总线和全局总线，其中局部总线分为纵横两个方向，全局总线一般贯穿整个 PE 阵列。

此外，互联结构还必须考虑线的数据位宽，对于不同的上层应用来说应该配置不同的数据位宽，这涉及 3 方面的问题：

- 数据通路的位宽。
- 配置资源的利用率。
- 灵活性。

4) 可重构 PE 单元和 PE 阵列的发展趋势

① 颗粒度由细变粗：主流的颗粒度为 8 比特、16 比特甚至是 32 比特。例如，美国加州大学 Irvine 分校的 Morphsys（1999）和 HP 公司的 Chess（1999）的 PE 均为 4 比特；卡内基梅隆大学的 Piperench（2000）、达姆施塔特大学的 DreAM（2001）使用 8 比特的 PE；ADRES（2003）处理器的颗粒度为 32 比特。颗粒度的增加可以显著减少配置信息的大小，以提高动态配置的速度。

② 阵列的规模不断增加：从原来的 4×4 到 8×8，再发展到上百个之多。例如，剑桥大学设计的

MATRIX (1996) 为 5×5 的 PE 阵列; ADRES (2003) 阵列大小为 6×6 ; XPP-III (2006) 包含了一个 6×8 的复杂阵列; DAPDNA-2 (2006) 中可编程单元的数目为 376 个。由于规模的增加, 整个可重构阵列的计算能力变得越来越强大, 可以容纳越来越复杂的运算流程。

③ 从同构到异构变化: PE 可以是 ALU、RAM、专用的运算块 (如乘法器), 甚至是一个 RISC 核本身。例如, MATRIX (1996) 的 ALU 均为一种仅能处理简单逻辑运算和加减功能的运算结构, Morphsys (1999) 的 PE 单元也是同样; ADRES (2003) 处理器包括了 RC 和 FU 两种 PE, XPP-III (2006) 内核中集成了 RAM-PAE、ALU-PAE 和 FNC-PAE 3 种 PE, 其中 FNC-PAE 是完整的 VLIW 型顺序处理内核。PE 的异构化, 使用 PE 阵列的计算能力不断加强。

④ 互联在可重构阵列的比重越来越大: 互联从一维到二维, 从单纯的 SwitchBox 和 ConnectBox 到现在复杂的 Router。例如, 华盛顿大学设计的 PaPiD (1996) 是一维互联的结构, 布线简单, 加州大学伯克利分校的 Garp (1997) 也采用同样结构。此后, 二维的互联结构变成主流。Morphsys (1999)、Chess (1999) 和 Piperench (2000) 均为二维机构; DAPDNA-2 (2006) 提出的动态网络结构可以提供 16Gps 的吞吐率, XPP-III (2006) 就有专门设计的 RAM I/O 总线和 FNC I/O 总线。可见, 互联的灵活性越来越高, 使得并行计算的规模可以进一步加大。

可重构 PE 的异构化、复杂化和 PE 阵列的规模化是今后的发展方向。互联灵活性和配置难度的折中会随着规模和单元复杂度的进一步增加而变得更加难以权衡, 是制约芯片性能同比增长的主要因素。重构单元的接口标准、阵列通信、同步的规范有待建立。以此规范、促进编译软件的开发, 加快应用移植, 促进可重构芯片的普及使用。

4. 可重构计算算法映射

可重构计算算法映射是在可重构计算架构的编程模型上将媒体算法映射到可重构计算架构的过程。

可重构计算架构是一种由配置信息决定功能, 由数据驱动的并行计算系统平台, 可重构计算架构实现算法时, 需要经过软硬件划分、子算法分割、DFG 图转换、阵列映射等步骤。

① 在软硬件划分过程中, 算法被划分成控制密集型任务和数据计算密集型任务, 因为可重构单元的特性只适合于计算密集型任务, 所以将控制密集型任务经过源码转换, 转换成 ARM 的源码交由 ARM 主控核来处理, 而数据密集型计算任务交由可重构单元来处理。

② 子算法分割指的是将交由可重构单元处理的算法源码进行分割, 分解成若干个子算法, 理由主要有两方面: ① 算法可能比较复杂, 而在一个阵列上不能一次性全部完成所有功能, 需要将整体的算法划分成若干个小的子算法, 复用硬件资源来完成算法。② 如软件代码中的子函数一样, 有部分核心子算法在整个算法中会反复使用, 将其单独提取出来, 可以使得配置信息的利用效率更高, 而不用重复性地存储相同功能的配置, 即算法的共性提取, 这在实现同一类型的多种算法时有很明显的优势。

③ 数据流图 (DFG, Data Flow Graph) 转换指的是将子算法分割后产生的子算法转换成 DFG 图, 因为可重构单元阵列只能识别配置信息, 且受到阵列的形状和资源数目的限制, 不可能直接将子算法代码交由可重构阵列执行, 因此需要经过并行化处理, 并根据可重构阵列结构形状等特性进行了优化之后再转换成与可重构阵列相对应的 DFG 图, 以提高阵列的使用效率, 提升性能。

④ 阵列映射指的是由配置工具将 DFG 图转换中所产生的子算法的 DFG 图转换成配置信息, 预先保存到存储器中。这个步骤类似于格式转换, 从可读的 DFG 图图形格式转换成可重构阵列能识别的配置信息格式, 总体的含义不变。

经过了这些步骤之后, 存储器中已经存放好了与算法相应的配置信息。在算法实现的时候, 由 ARM 主控核根据划分给它的算法任务的执行来选择和判断可重构阵列中将要执行的功能, 而从预先保

存的配置信息中选取相应的配置信息对可重构单元进行初始化，可重构单元在用配置信息初始化之后，就会具有某种特定的功能，能完成某个特定的子算法；向可重构单元输入相应的待处理数据，则由数据驱动的可重构单元会在配置信息的指引下完成相应的操作。在完成一个子算法之后，主控核会根据算法判断和选取另一个子算法对可重构单元进行初始化，执行下一个子算法功能。最终，经过了多个子算法的功能组合，实现一个完整的算法。

5. 可重构计算任务编译

任务编译技术以及任务编译器设计是可重构计算技术研究的核心之一，国际知名的几款可重构处理器都有与其对应的任务编译器。美国加州大学 Irvine 分校设计的 Morphosys 处理器采用 MCC 作为其编译器，MCC 是专门针对 Morphosys 硬件结构设计的编译工具。MCC 要求输入的源代码中必须指明在可重构阵列中执行的函数块，这等同于手工进行软硬件划分，自动化程度低。加州伯克利分校为其研发的 Garp 处理器专门设计了 GarpCC 编译器，该编译器针对 Garp 处理器的可重构硬件特性实现计算任务软硬件自动划分、配置信息自动生成，并以 GNU GCC 编译器为平台，实现了从 C 源程序到可执行代码的完整编译流程。GarpCC 选取的加速目标是 C 源代码中的循环体，针对每个循环体进行算子匹配、自动生成配置信息，该配置信息以静态数组的形式存在于 C 源代码中，从而保证了程序的 C 语言兼容性，因而最终采用 GCC 编译器即可完成可执行代码的生成。在另外一些可重构处理器设计中，研究者可为可重构处理器开发了全新的编程语言，如 SA-C、Silver-C 语言等。SA-C 语言是面向表达式的编程语言，在语句中就能很好地表达数据并行操作。然而，SA-C 语言不能兼容传统的高级编程语言，这势必造成大量已有的软件成果无法被重用和继承。Silver-C 也面临着同样的问题。

纵观任务编译技术的发展，可以归纳出其趋势是向着全自动化、高兼容性的方向发展。全自动化即任务编译器只需要输入应用源程序，无需人工干预地自动完成软硬件划分、计算任务调度和映射、配置信息生成、可执行代码生成等工作；高兼容性是指能够处理已被广泛应用的程序设计语言，如 C/C++ 等，无需或只需很少的对源代码的改动，就可用任务编译器生成高性能的可重构处理器执行代码。

案例：REMUS-II 粗粒度可重构计算架构

1. REMUS-II 基本架构

通常，一个粗粒度可重构系统由一个主处理器、若干个粗粒度可重构阵列、配置信息管理单元、片上与片外存储器、相关总线和接口等组成。REMUS-II 是一款面向高性能媒体处理应用的粗粒度可重构系统，其基本架构如图 4-62 所示。REMUS-II 主要包含主控 ARM 核（使用 ARM7TDMI 处理器）、便签式内存（SPM Scratch-Pad Memory）、直接内存访问控制器（DMAC，Direct Memory Access Controller）、中断控制器、两个可重构处理单元（RPU，Reconfigurable Processor Unit）、微处理器单元（ μ PU，Micro-Processor Unit）、外部存储器接口（EMI，External Memory Interface）等功能模块。外部存储器使用容量为 16MB 的 DDR SDRAM 存储器，用于存放媒体解码过程中的大量计算数据，如视频码流及参考帧数据等。系统总线为 32 比特位宽的 AMBA 2.0 AHB 总线，用于连接片上的各个功能模块。主控 ARM 核通过 AHB 总线对各模块进行访问和调度，片上模块通过数据传输位宽为 64 比特的 EMI 来访问外存 DDR。

RPU 是 REMUS-II 可重构系统中主要的计算部件，用于完成媒体处理任务中的计算密集型部分。每个 RPU 主要由 4 个可重构单元阵列（RCA，Reconfigurable Cell Arrays）组成，每个 RCA 中包含一个 8×8 的 PE 阵列。PE 的核心部件是 ALU，其计算位宽为 16 比特，支持多种算术及逻辑运算，整个

REMUS-II 中总共有 512 个 PE。配置信息接口（CI，Configuration Interface）负责接收及缓存 μ PU 发送至 RPU 的配置信息，而数据交换缓存（DEB，Data Exchange Buffer）用于进行两个 RPU 之间的数据传输。

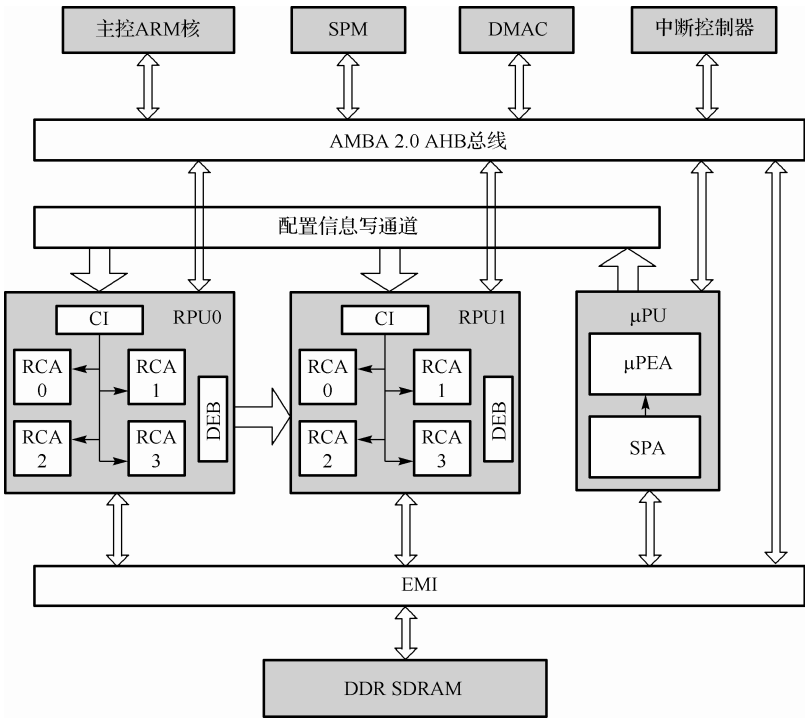


图 4-62 REMUS-II 可重构系统基本架构

μ PU 是 REMUS-II 可重构系统中的核心控制部件，负责生成配置包（CP，Configuration Package）及进行不合适 RPU 完成的控制密集型操作。 μ PU 由微处理器内核阵列（ μ PEA，Micro-Processor Element Array）和流处理器阵列（SPA，Stream Processor Array）构成，其中 μ PEA 包含 8 个 ARM7 微处理器，其功能是将 SPA 输出的控制信息打包成配置包，然后输出到 RPU 中。

2. REMUS-II 配置体系

1) 配置信息分层机制

由于 REMUS-II 系统规模庞大，计算及互联资源丰富，导致其配置信息的数据量也很大。为了减轻配置信息存储及传输的压力，RPU 的配置信息采用了分层机制，具有 3 个配置信息层级（CL，Configuration Level），从高到低依次为 CL0 层、CL1 层、CL2 层。

CL0 是最高层的配置信息，也称为配置包，用于处理 RPU 与片外之间的数据交互以及调用 CL1 层配置信息。一套配置包的长度为若干个 word（1word=32 比特），其中第一个 word 为包头，标明了所要配置 RCA 的编号，第二个 word 为配置字（CW，Configuration Word），标明了所调用 CL1 层配置信息的索引，剩余的 word 为立即数及外部数据读写模块的配置信息。

CL1 是中间层的配置信息，也称为配置组（CG，Context Group），用于处理 RPU 内部和 RCA 之间的数据交互以及调用 CL2 层配置信息。一套配置组的长度为 5~73word，其中第一个 word 为配置组头，标明了本套配置组的长度（以 word 为单位）及所调用 CL2 层配置信息的个数（1~16 个），而在随后的 word 中会依次标明这些 CL2 层配置信息的索引。

CL2 是最低层的配置信息，也称为阵列配置（CC，Core Context），是各种计算密集型算法数据流图在 PE 阵列上的直接映射。一套阵列配置由 107 个 word 构成，主要用于配置各个 PE 的输入、输出、运算类型，以及 PE 之间的互联关系。

配置信息的分层机制，其实是根据 RPU 的计算资源规模和硬件层次结构，对应用算法进行合理划分的结果。一般来说，CL0 指示实现某一个子算法的 RCA 编号并对阵列与片外间的数据交互进行配置，CL1 对应于该子算法，能够完成一个相对独立的功能，CL2 则对应于该子算法中的一个子函数，一般是重复多次执行的计算任务，完成一个较为简单的功能。图 4-63 以媒体解码应用中常用的运动补偿算法为例，说明了层次化配置信息与应用算法的对应关系。在完成 luma_16x16_mc00 的功能时，由于 RCA 阵列规模的限制，需将其划分到 4 个 RCA 上并行执行，每个 RCA 分别调用 luma_8x8_mc00 子算法。而在 luma_8x8_mc01 和 luma_8x8_mc02 子算法中，都调用了 func_C 函数，因此 CL1_1 和 CL1_2 都调用了 CL2_2 配置信息。由此可见，一套高层配置信息可以调用多套低层配置信息（CL0 层除外，一套 CL0 层配置信息只能调用一套 CL1 层配置信息），而一套低层配置信息也可以被多套高层配置信息调用。因此，这种配置信息分层机制能够最大限度地减少配置信息的总大小，节省配置信息的片上存储开销。

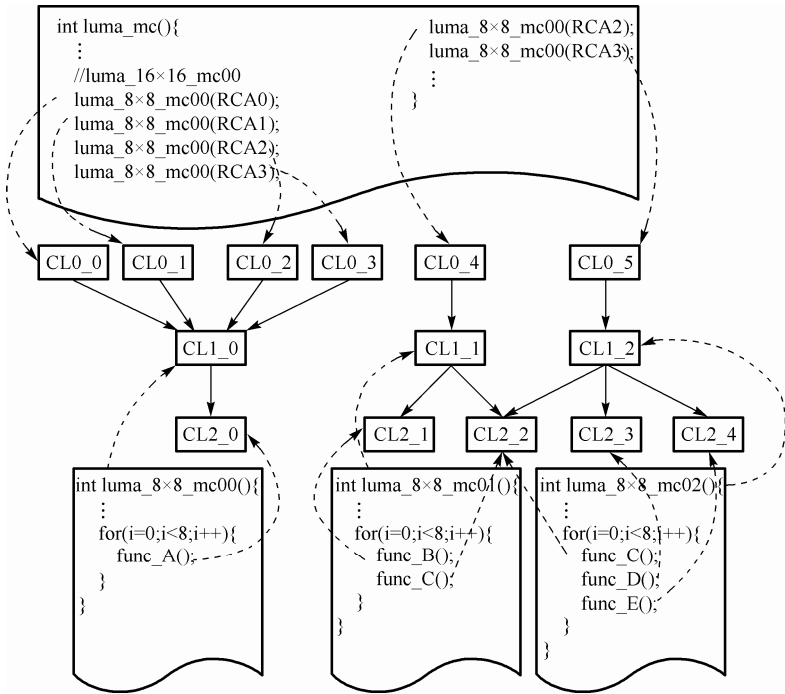


图 4-63 各层配置信息与运动补偿算法的对应关系

2) 配置流控制子系统

在 REMUS-II 中，RPU0 和 RPU1 的配置信息是独立管理的，每个 RPU 都有一套相同的配置流控制子系统（CFC，Context Flow Controller），其结构如图 4-64 所示。需要注意的是 RCA 层的模块共有 4 组，图中只标出了 1 组。CI 中有两个子模块：配置信息初始化接口（CII，Context Initialization Interface）负责对片上的配置信息存储器进行初始化，配置字接口（CWI，Configuration Word Interface）用于提取和传输 CL0 层配置信息中的配置字。配置信息存储器包括 RPU 层的全局配置组存储器（GCGM，Global Context Group Memory）、全局阵列配置存储器（GCCM，Global Core Context Memory），以及

RCA 层的局部配置组存储器（LCGM，Local Context Group Memory）。其中，GCGM 和 GCCM 存储某一种应用算法中所有的配置组和阵列配置，LCGM 作为配置传输过程中的一个缓存，存储一套配置组。表 4-25 给出了各配置信息存储器的容量及传输位宽。配置信息传输控制模块包括用于读取并解析配置信息的 RPU 配置解析器（RPU Context Parser）和 RCA 配置解析器（RCA Context Parser），以及用于控制配置信息读写操作并发出中断信号的 RPU 控制器和 RCA 控制器。

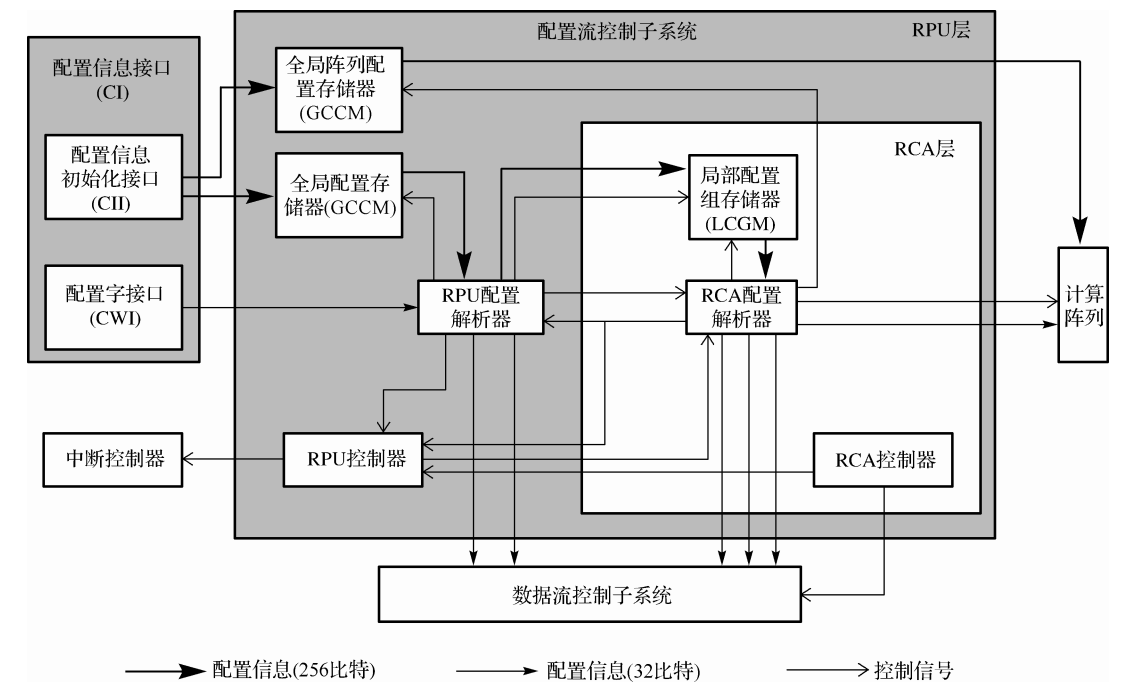


图 4-64 配置流控制子系统结构框图

表 4-25 各配置信息存储器的容量及传输位宽

配置信息存储器	宽度（比特）	深度	容量（KB）	数量	总容量（KB）	传输位宽（比特）
GCGM	256	512	16	2	32	256
LCGM	256	16	0.5	8	4	256
GCCM	1024	1024	128	2	256	1024

3. REMUS-II 可重构系统的重构过程

REMUS-II 为动态可重构系统，某个 RCA 在进行重构（包括配置信息的传输及阵列功能的改变）的同时，另一个 RCA 可以不受干扰地执行计算任务。REMUS-II 系统的重构过程分为 4 个步骤，具体描述如下。

- （1）配置信息初始化：在系统执行应用算法之前，通过 CII 将所需的所有配置组和阵列配置初始化到 GCGM 和 GCCM 中。
- （2）配置包传输：μPU 中的 μPEA 将 SPA 输出的控制信息打包成配置包，通过配置信息写通道将其传到 CWI，缓存在配置包 FIFO 中。
- （3）配置组传输：RPU 配置解析器从配置包 FIFO 中读取配置包并将其分解，从配置字中解析出所需配置组在 GCGM 中的地址，随后从 GCGM 中读取该套配置组并存入相应 RCA 的 LCGM 中；与此同时向 RPU 的外部数据读写模块传输相应的配置信息。

（4）阵列配置传输：RCA 配置解析器从 LCGM 中读取配置组，解析出所需若干套阵列配置在 GCCM 中的地址，随后从 GCCM 中依次读取各套阵列配置并发送到 PE 阵列，从而改变 PE 阵列的功能；与此同时向 RCA 内部的各个数据处理模块传输相应的配置信息。

以上 4 个步骤中，除了第（1）步只在系统初始化时进行一次，其余的 3 个步骤在整个系统运行过程中是流水化进行的。图 4-65 表明了 REMUS-II 系统流水化的重构过程，图中不同的编号代表不同的配置包。一般来说，由于一套配置组中会调用多套阵列配置，因此 RCA 配置解析器进行阵列配置传输的时间相对较长，一套配置包对应的重构时间基本由阵列配置的传输时间决定。另外，配置组中标明了各套阵列配置的循环次数，这样 PE 阵列能够实现“一次功能配置、多次循环计算”，减少了从 GCCM 中读取阵列配置的次数，进一步缩短了配置信息的传输时间。

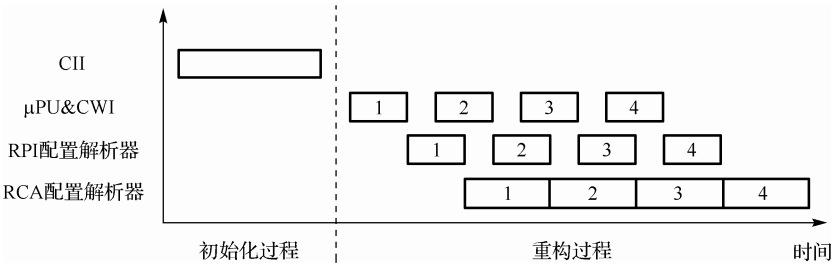


图 4-65 REMUS-II 系统的重构过程

4. 媒体应用在 REMUS-II 可重构计算架构上的实现

REMUS-II 可重构系统主要面向的是以高清视频解码为代表的媒体应用，支持包括 H.264、MPEG-2 在内的多种主流视频编解码格式。虽然这些标准有所不同，但都采用了预测加变换的混合编码模型，其解码流程及核心子算法基本都是相同的，主要的不同在于各功能块的细节。图 4-66 是 H.264 算法的解码框图，主要有熵解码、反量化、反变换、帧内预测、运动补偿、去块效应滤波等几个子任务。其中，熵解码为控制密集型任务，由 μPU 中的 SPA 完成，其余的计算密集型任务均由两个 RPU 完成。

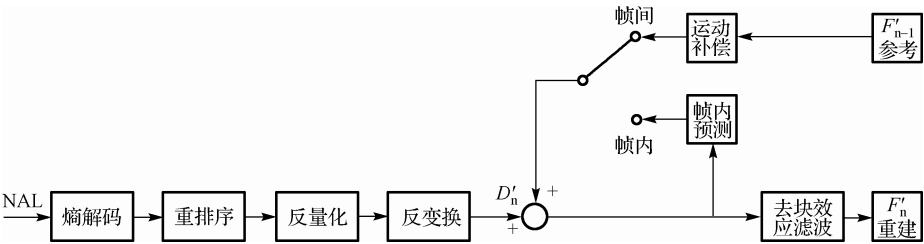


图 4-66 H.264 算法的解码框图

视频解码任务在 REMUS-II 可重构系统上的实现可划分为熵解码、重构过程、计算过程 3 部分，以 3 级流水的方式进行，其中熵解码以条带（slice）为单位，重构过程和计算过程都以宏块（MB，Macro Block）为单位，一个宏块的大小为 16×16 像素。RPU 有两种计算模式：当计算任务较为简单、花费时间较少时，两个 RPU 可以执行相同的任务，称为并行模式；当计算任务较为复杂、花费时间较多时，两个 RPU 可以以流水的方式执行不同的任务，称为乒乓模式。由于 MPEG-2 计算复杂度较小，因此采用并行模式；而 H.264 算法复杂度较高，由一个 RPU 单独完成所有的计算任务耗时较长，难以达到高性能要求，因此采用乒乓模式，其中 RPU0 执行反变换、帧内预测、运动补偿等子任务，RPU1 执行去块效应滤波的子任务。两种解码算法在 REMUS-II 上的实现方式如图 4-67 所示，图中 Tmb 代

表解码一个宏块所需的时间， T_{mb} 越小，解码性能越高。表 4-26 给出了 H.264 和 MPEG-2 算法配置信息的相关情况，其中 H.264_pred 和 H.264_deb 分别代表 H.264 算法在 RPU0 和 RPU1 上执行的部分。从中可以看出，由于算法复杂度较大，H.264 的配置信息比 MPEG-2 要大得多；而在 H.264 的配置信息中，H.264_pred 又占了很大一部分。

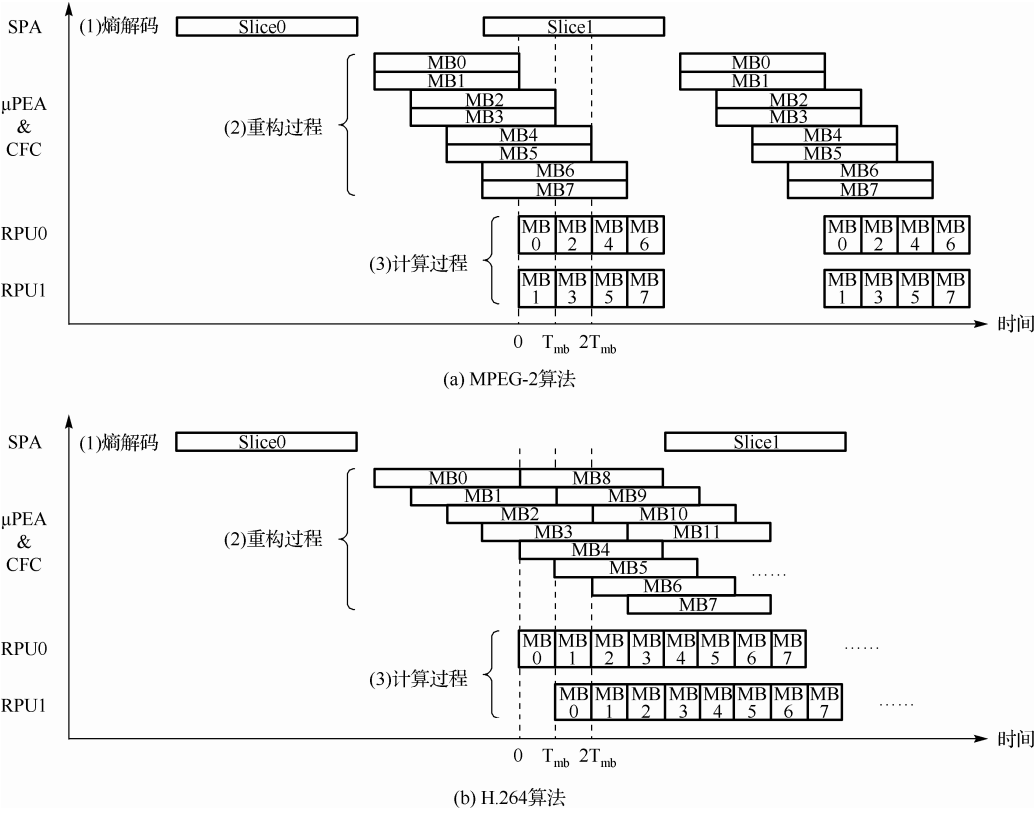


图 4-67 MPEG-2 和 H.264 算法在 REMUS-II 上的实现方式

表 4-26 H.264 和 MPEG-2 算法配置信息的相关情况

算法	配置组				阵列配置			
	套数	字数	总大小 (KB)	占 GCCM 比例	套数	字数	总大小 (KB)	占 GCCM 比例
H.264_pred	305	3582	13.4	87.45%	197	25216	98.5	76.95%
H.264_deb	34	848	3.3	20.70%	13	1664	6.5	5.08%
MPEG-2	21	256	1.0	6.25%	15	1920	7.5	5.86%

正是基于这种 3 级流水的实现方式，系统较长的重构时间被隐藏起来，从而达到了较高的重构效率。在 200MHz 的系统时钟频率下，REMUS-II 支持 H.264 High Profile @ Level 4 格式的视频解码，性能能够达到 1920×1080@30fps；支持 MPEG-2 Main Profile @ High Level 格式的视频解码，性能能够达到 1920×1080@40fps。

思考题

- 1. 给出几个影响 CPU IPC 的例子，并给出可能的解决方案。

2. 对于一个简单的 2 比特分支预测器，如果一共采用 4096 个表项，那么这个分支预测器的 BPB 一共要占用多少存储空间？需要用多少位 PC 值作为索引？分别是第几位到第几位？（提示：假设 PC 的值总是最低两位为 0，因为指令只能存放在 4 字节对齐的地址。）
3. 除了我们介绍的 RAW、WAR 和 WAW 3 类数据冒险外，是否存在 RAR（读后读）冒险？为什么？
4. 请讨论为什么 ARM 指令集中除了支持 SUB 减法指令外，还需要支持 RSB 反向减法指令？为什么没有反向加法指令？

扩展阅读

- [1] John L. Hennessy, David A. Patterson 著. 计算机体系结构量化研究方法（第 5 版）[M]. 贾洪峰译. 北京：人民邮电出版社，2013.
- [2] Jean-Loup Baer. Microprocessor Architecture - FROM SIMPLE PIPELINES TO CHIP MULTIPROCESSORS[M], Cambridge University Press, 2010.
- [3] ARM. ARM11 MPCore Processor Technical Reference Manual[Z].2008-10-15.
- [4] Smith, J. E., & Sohi, G. S. The microarchitecture of superscalar processors[J]. Proceedings of the IEEE. 1995.
- [5] Patt, Y. Requirements, bottlenecks, and good fortune: agents for microprocessor evolution[J]. Proceedings of the IEEE. 2001.
- [6] Kessler, Richard E. The alpha 21264 microprocessor[J]. Micro, IEEE 19, no. 2 (1999): 24-36.
- [7] 冯子军, 肖俊华, 章隆兵. 处理器分支预测研究的历史和现状[J]. 信息技术快报, 6, no. 4 (2008): 21-25.
- [8] 杜春雷. ARM 体系结构与编程. 北京：清华大学出版社，2003.
- [9] Joseph Yiu. ARM Cortex-M3 权威指南. 宋岩译. 北京：北京航空航天大学出版社，2009.
- [10] Dominic Sweetman. MIPS 体系结构透视. 李鹏，鲍峥，石洋等译. 北京：机械工业出版社，2008.
- [11] David A. Patterson John L. Hennessy. 计算机组成与设计——硬件/软件接口. 康继昌 樊晓桢 安建峰等译. 北京：机械工业出版社，2011.
- [12] ARM Architecture Version 6 (ARMv6) [EB/OL]. <http://www.arm.com>, 2002.
- [13] ARM Ltd. ARM Processor Core Overview [DB/OL]. <http://www.arm.com/products/CPU/>
- [14] ARM1176JZF-S Technical Reference Manual [DB/OL]. http://www.arm.com/pdfs/DDI0301F_arm1176jzfs_r0p6_ttm.pdf
- [15] ARM9E-S Core Technical Reference Manual[DB/OL]. <http://infocenter.arm.com>, 2004.
- [16] Knuth,D.E.The Art of Computer Programming[M],Vol.3,2nd ed.Sorting and Searching.1998.
- [17] MIPS R4000 Microprocessor User's Manual[DB/OL] http://www.ece.cmu.edu/~ece447/s15/lib/exe/fetch.php?media=mips_r4000_users_manual.pdf.
- [18] 北京龙芯中科技术服务有限公司.龙芯 2F 处理器用户手册. Revision1.0[EB/OL]，2008.
- [19] 北大众志. Book3-UniCore-2 处理器结构手册-v2.12[EB/OL] 1-2,2011.

第5章 存储子系统

5.1 存储子系统概述

嵌入式系统中，存储系统是一个重要的组成部分。在片上系统的设计过程中，存储子系统制约了整个片上系统的功耗、性能和芯片面积，成为片上系统的核心设计部分。在实际芯片应用过程中，存储器是被访问最多的模块，取指令/数据或存放变量等操作无时不在进行，这些都需要用到存储器。因此，存储器的访问速度以及因为这些访问所带来的功耗成为了片上系统性能优化和功耗降低的主要研究方向。自20世纪80年代以来，处理器速度与存储器访问速度的差距不断增大，处理器的速度以每年50%~100%的比例增长，而作为片外主存储器的DRAM的速度增长则一直维持在每年7%左右。这种处理器与存储器速度的差值一般也被称为存储墙（Memory Wall），如图5-1所示。存储墙的存在日益成为嵌入式系统性能、功耗和成本的瓶颈，如何优化存储子系统的架构及管理策略，一直是SoC研究的关键。事实上，在其最新出版的关于存储子系统的煌煌巨著 *Memory Systems: Cache, DRAM and Disk* 《存储器系统，Cache，DRAM与磁盘》一书中，Bruce Jacob等人指出“存储子系统的设计已经成为当前以及最近若干年来微处理器和系统设计过程中唯一重要的设计因素”。（参见扩展阅读[1]的前言。）

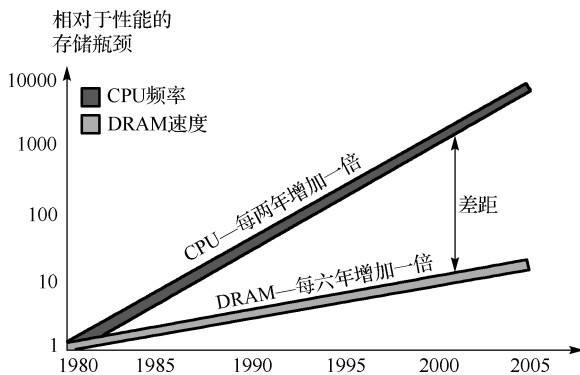


图 5-1 处理器-存储器性能差

存储系统是那些用来长时间存储大量数据信息的计算设备的集合。它通常包含多种存储介质，这些介质能在成本、性能、可靠性、存储密度以及功耗方面提供不同程度的折中。存储系统包括用于存储数据信息的硬件设备，用于数据信息传递的媒介，以及对硬件进行控制、对数据进行组织管理的软硬件模块。由于需要较大的存储容量与较低的单位存储价格，因而通常希望使用大容量存储提供的存储器技术（大容量存储器价格便宜、存取时间较慢）；但为达到性能要求，又需要使用昂贵的、容量相对较小而具有快速存取时间的存储器。解决这一矛盾的办法是：不依赖于单一的存储器部件或技术，而是使用层次化的存储器结构。

图5-2详细示出了偏重于SoC角度存储子系统的层次性结构，从图中可以看出，沿着存储器层次结构自顶向下，每层存储器的单位成本降低，容量变大，存取时间变长，访问频度降低，访问功耗变大。

如图 5-2 所示，寄存器处在最顶层，位于处理器内核中，在存储子系统中提供最快的存储访问速度。例如，ARM7TDMI 内部就含有 37 个 32 位寄存器，共 148 字节。通常情况下，存储器越接近处理器，其处理速度越快，容量也越小；反之，越远离处理器，其处理速度越慢，但容量也越大。

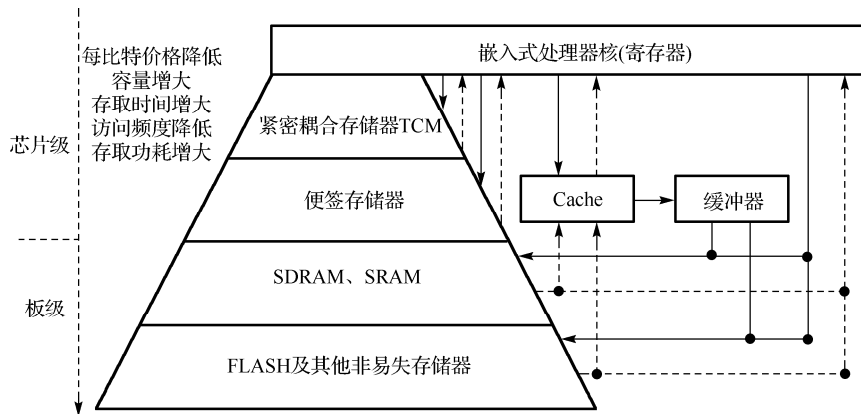


图 5-2 存储子系统详细层次结构

接下来的一层仍然在芯片内部，通常包含紧密耦合存储器 TCM 和片上 SRAM（在本书中，如不特殊说明，片上 SRAM 就是指便签存储器 SPM，Scratch-pad Memory），或者是片上 Cache，通常只有几百个 B 至几十 KB，它们和 CPU 内核通过专用的片上总线连接。在一些 CPU 设计中带有专用 SRAM，与一般存储空间相同，CPU 直接访问，无需 Cache 结构中的控制逻辑。此类存储空间称为 RAM 缓存（Cache as RAM），由于其与 CPU 紧耦合，传输延时较低，又被称为紧耦合存储器（TCM，Tightly Coupled Memory）。现代高性能微处理器往往包含多级缓存系统，通常第一级（L1）Cache 和第二级（L2）Cache 集成在片上，L3 Cache（如果有的话）通常是作为一个单独的硅片与处理器封装在一起。

再接着往下就是板级存储器（或称为片外存储器），大致可以分为主存和掉电非易失存储器两层，前者通常是由同步动态随机存取存储器（SDRAM）构成，通常可达数百 MB 或是数 GB，在考虑性能因素并且所需存储器容量不大时，也可以采用 SRAM；后者（掉电非易失存储器）通常采用 Flash 或者微硬盘，有的甚至还包含外存储卡，可达数 GB 或是数十 GB。通过在系统中加入适当的存储器管理，程序大小就不再受主存储器的限制，而取决于这些掉电非易失存储器的容量。采用这种层次性结构（配合 CPU 访存的时间局部性和空间局部性），就可以用低速、高功耗存储器的平均价位得到高速、低功耗存储器的性能，并满足嵌入式系统对存储器容量的需求。

一般来说，绝大多数应用通常只会以较高的频率访问存储器中一个相对较小的局部存储区，而偶尔访问存储器的其他部分。只要将频繁访问的部分置入相对较为顶层的存储器，就可以获得较高的访问性能。这也是设计层次性存储结构的初衷。

5.2 高速缓存 Cache

5.2.1 Cache 的基本组成

Cache 由高速 SRAM 组成。SRAM 的全称是静态随机存储器（Static Random Access Memory），每比特由 6 个晶体管构成，以双稳态电路形式存储数据，不需要刷新电路即能保存内部数据。与动态随

机存储器 (Dynamic Random Access Memory) 相比, SRAM 结构复杂、制造成本高, 但其读写性能也较高, 且能耗较低, 并且其制造工艺与普通 CMOS 工艺兼容。因此 SRAM 主要用在比主存小得多的片上存储器中, 如 SoC 中的 Cache 和片上 SPM 都使用 SRAM。但是 SRAM 存储的数据在掉电后会丢失。(关于 Cache 的实现细节, 感兴趣的读者可以参阅扩展阅读[1]第一部分的第五章。)

5.2.2 Cache 的基本原理

1. Cache 的作用

Cache 是比主存容量小得多的高速 SRAM 单元，一般价格较昂贵，其速度接近 CPU 速度（即接近单 CPU 时钟周期的读写操作）。当 CPU 访存时，如果存取单元的内容在 Cache 中，就对 Cache 进行访问，称为 Cache 命中；如果存取单元的内容不在 Cache 中，就从主存中把相应块中的指令或数据调入 Cache，然后再进行访问，称为 Cache 不命中，或 Cache 缺失。Cache 的目的是在系统中提供一个较大且较便宜的存储器的同时可以使存取数据的速度接近 CPU 的速度，使 CPU 的性能得到充分发挥。因此从 CPU 的角度看，Cache 的出现可以解决容量、速度与成本之间的矛盾。

Cache 的所有功能都是用硬件来实现的，因此它对于应用程序员和系统程序员来说都是透明的^①。行（Cache line）是 Cache 与主存之间进行数据交换的单位。CPU 在读取指令或者数据的过程中会同时将要读取的及其相邻的指令或者数据保存到一个 Cache 行中，这样当 CPU 再次需要读取这些指令或数据时，就可以从 Cache 行中直接得到相应的指令或数据。由于 Cache 的存取速度与 CPU 速度相当，这样对于整个系统而言，性能会得到很大的改进。事实上，在 CPU 访问一个数据之后，在不久的将来它极有可能访问相同的数据，这种规律被称为**时间局部性**。

对于不同的系统而言，Cache 行的大小也不同，一般来说 Cache 行大小为几个字，视 Cache 的容量而定。当 CPU 从主存中读取一个字的数据时，该字和该字相邻的内容都会被读取到同一个 Cache 行中。比如，如果 Cache 行的大小为 4 个字，当某个地址为 $N+1$ 中的内容被读取时，Cache 的控制逻辑会同时将地址 N 、 $N+1$ 、 $N+2$ 、 $N+3$ 中总共 4 个字的内容读取到同一个 Cache 行中。如此，当地址 N 、 $N+2$ 或者 $N+3$ 中的内容在这之后被 CPU 读取时，所访问内容就可以直接从 Cache 中得到，访存的速度将得到很大提高。事实上，在一段时间内 CPU 访问相邻存储空间的概率是很大的，这种规律称为**空间局部性**。

一般而言，主存由多达 2^n 个字组成，这些字可以被寻址并且每个字都有唯一的 n 位地址。为实现从主存到 Cache 的映射，主存被看成由许多定长的行（这个行的容量与 Cache 的行容量相等）组成，每行有 K 个字，共有 $M=2^n/K$ 个行。对应的 Cache 则由 C 个行组成，每行也有 K 个字。Cache 中行的数量远远小于主存中行的数量（ $C \ll M$ ）。在任一时刻，主存中都有若干个行的副本被保存在 Cache 中。当 CPU 读取主存中某行中的某字时，该字所在的整行内容被传送到一个 Cache 行中。由于主存的行数多于 Cache 的行数，所以单个 Cache 的行不能一直被主存中的一个行专用，也就是说 Cache 中的某个特定行可能被映射到多个不同的主存行。每个 Cache 行都有一个标签（Tag）存储器，其中存放了该行中所保存的主存储器行的部分地址和一些控制信息，用于识别主存的哪个行被映射到了当前的 Cache 行中。

2. Cache 的基本架构

一般来说，Cache 按其组织形式大体上可分为两种：一种是单一型的 Cache 结构，指令和数据被统一存放在一个 Cache 中，这种结构又称为普林斯顿（Princeton）结构。另外一种分离式的 Cache 结构，指令和数据分开存放在两个 Cache 中，这种结构又称为哈佛（Harvard）结构。

采用哈佛结构的 Cache 是因为它具有一些比较明显的优点。首先，指令 Cache 和数据 Cache 能同时为 CPU 提供指令和数据，使得取指令的操作和存取数据的操作能同时进行（在流水线处理器中，确

^① 这里所说的“透明”是指程序员感觉不到 Cache 的存在，也无法操纵 Cache 的行为。当然，在系统初始化时，系统程序员需要对 Cache 模块进行必要的配置，并使能 Cache。

实会出现某条指令处于取指阶段，而另一条指令同时处于存储器访问阶段)。其次，哈佛结构允许设计者独立地选择和优化指令 Cache 和数据 Cache 的容量、关联度和 Cache 行的大小，使得指令 Cache 和数据 Cache 的配置更加符合应用的需要。

哈佛结构的不足之处在于，当不同的程序对于指令 Cache 和数据 Cache 的容量有不同的要求时，哈佛结构的 Cache 无法改变分配，而普林斯顿结构的 Cache 却能自动保持这种动态平衡，使指令 Cache 和数据 Cache 的分配达到最优。

尽管两种 Cache 各有优缺点，但就发展趋势而言，分离式结构的 Cache 如日中天，远远超过了单一式的 Cache。现在越来越多的超标量 CPU 在第一级都采用分离式的 Cache，这些超标量 CPU 强调指令的并行执行，并采用一种方法预取未来执行的指令。分离式 Cache 在基于指令流水线的任何设计中都是重要的，因为它消除了 Cache 在一条指令取指和另一条指令访存时的结构冒险。假设指令流水线的的第一级是取指，而第四级是访存，这样，当某一条指令执行到第四级向 Cache 发出数据请求时，这条指令后的第三条指令正好处于取指级，会向 Cache 发出指令请求。此时，如果是单一式 Cache，指令请求将被暂时阻塞。而分离式 Cache 将不会出现这个问题，指令请求和数据请求将同时被处理。由于上述问题在程序的执行过程中普遍存在，因此单一式 Cache 性能较低，这也是分离式 Cache 被广泛应用的重要原因之一。在现代微处理器设计中，往往第一级 Cache 采用分离的 Cache 架构，而第二级乃至第三级 Cache 都采用单一的 Cache 架构。

3. Cache 的映射方式

由于主存的行数远远多于 Cache 的行数，因此需要一种算法把多个主存的行映射到同一个 Cache 行中。并且，还需用一种方法来确定主存中的哪一个行当前占用了 Cache 行。选择哪种映射方法决定了 Cache 的不同结构，通常有 3 种映射方法：直接映射、组关联映射和全关联映射。

(1) 直接映射 (Direct Mapping)。

直接映射是最简单的映射方法，每个固定可用的 Cache 行都可以被一组具有共同特征的主存行所映射，而这组主存行中的每一个都只能映射到该 Cache 行中，不能映射到其他任一 Cache 行中（即使其他 Cache 行是空的）。直接映射的技术实现起来比较简单，并且由于每次判断是否命中时地址的高位只与一个 Cache 行的 Tag 比较，因此功耗较低。它的主要缺点是对于某个确定的主存行，其在 Cache 中的位置固定。因此，如果两个可以映射到同一 Cache 行却来自不同主存行的字恰好被一个程序连续重复引用，那么这两个主存行将不断交替地进入 Cache 行中，Cache 的命中率将会大大降低（此时称 Cache 进入颠簸状态）。我们以图 5-4 所示的 8 位地址直接映射 Cache 为例来介绍直接映射 Cache 的原理。假设 CPU 的地址为 8 位（实际的地址可能是 32 位，甚至 64 位），共可以寻址 256 字节，假设 Cache 的容量为 64 字节，每行的大小为 8 字节，共 8 个 Cache 行。而对应的存储器空间可以分为 32 行。8 位地址被分为 3 部分，其中最低 3 位是行内偏移，用于表示一个行中的字节偏移量；然后是 3 位索引地址，索引地址实际上是 Cache 行的编号，3 位索引对应 8 个 Cache 行；最高两位是 Tag 地址。当 CPU 发出访存地址时，Cache 控制器首先利用 3 位索引地址找到相应的 Cache 行（包括数据存储器和 Tag 存储器），数据存储器输出该行的数据到 Mux，同时地址中的 Tag 位（这里是地址的最高两位）会与选中的 Tag 行进行比较。如果数据有效位置位（表明该行内数据有效），同时两个 Tag 相等，则输出 Cache 命中信号，行内偏移地址被送往 Mux 已选择读出行中的相应字节。直接映射的每个索引只对应于一个 Cache 行，因此只要地址的索引段相等，就会被映射到固定的 Cache 行。在本例中，一个 Cache 行可以被映射到 4 个不同的存储器行。

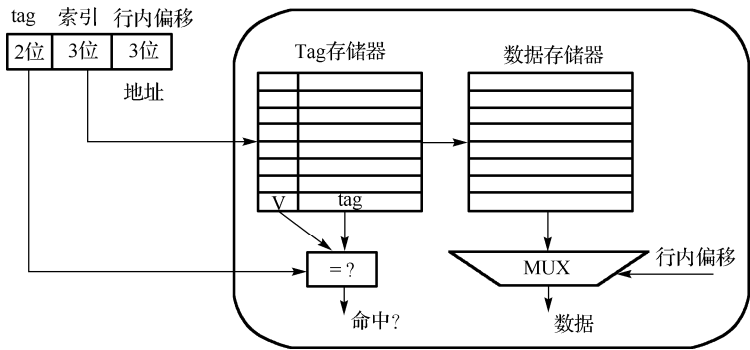


图 5-4 8 位地址的直接映射 Cache

(2) 全关联映射 (Fully Associated Mapping)。

为克服直接映射的缺点，全关联映射的原理得以出现并发展。它允许每个主存行可以填充到 Cache 的任意一行中（也就是说只要 Cache 中存在空行，就可以将需要映射的内存行装载到该 Cache 行）。在全关联映射中，Cache 的控制逻辑简单地把存储器地址分为高位的标记（Tag）域和低位的字（Word）域，Tag 域唯一标识主存行。为了确定 CPU 要存取的某个字是否在 Cache 中，Cache 的控制逻辑必须同时将 CPU 存取地址的高位（也就是 Tag 地址段）与每个 Cache 行中的 Tag 进行比较，判断其是否相等。对于全关联映射，某个确定的主存行可以进入 Cache 的任一行中，具有很大的灵活性。全关联映射的缺点是需要设计的电路复杂，并且每次命中判断需要检查所有 Cache 行的 Tag 域，功耗较大。这里继续用我们的 8 位地址 CPU 作为例子来说明全关联 Cache 的工作原理。如图 5-5 所示，我们的全关联 Cache 还是拥有 8 个 Cache 行，但与直接映射不同的是、8 位地址被分为两部分——Tag 地址和行内偏移。因为 Cache 的行大小没有改变，依然是 8 字节，因此行内偏移地址依然占据低 3 位，而剩下的高 5 位地址现在都变成了 Tag 地址（注意，在全关联 Cache 中没有索引地址了）。当 CPU 发出 8 位地址的访存操作时，高 5 位的 Tag 地址将直接被送往 Tag 存储器，与每一个 Cache 行的 Tag 地址进行比较，同时 8 个 Cache 行的内容被一起读出并送往第一级 Mux，如果 CPU 发出的 Tag 地址与某个 Cache 行的 Tag 地址相等，则表示该行命中，并将被选中的行编号送往第一级 Mux，用于选择被选中行的内容输出，然后通过第二级 Mux 和行内偏移地址选择需要读出的字节。显然，全关联 Cache 的硬件复杂度要比直接关联复杂得多。

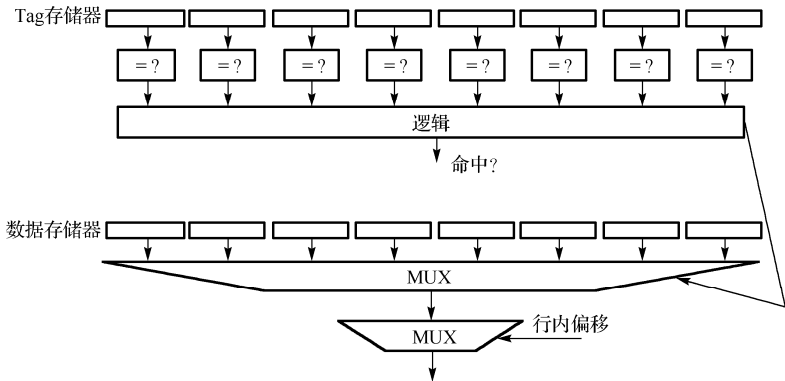


图 5-5 8 位地址的全关联 Cache

(3) 组关联映射 (Set-Associated Mapping)。

组关联映射既有直接映射和全关联映射的优点，又在一定程度上避免了它们的缺点，是上述两种

结构的一种折中办法。在组关联映射中，Cache 分为 v 个组，每组有 k 个行，称为 k 路组关联映射。对于 k 路组关联映射，某一个确定的主存行可映射到某一特定组（共 v 组，内存行所映射的组号由其地址中的索引段决定）中 k 个 Cache 行中的任意一行。（也就是说，一个特定内存行所映射的 Cache 组是固定的，但组内的行是可以随意映射的。）组关联映射具有较强的灵活性，而且每次命中判断时 CPU 发出的地址只需与所映射组中的 k 个 Tag 相比较，因此功耗较低。我们通过图 5-6 和图 5-7 说明组关联映射，对于 2 路组关联映射，8 位地址被分为 3 部分：3 位行内偏移、2 位索引地址和 3 位 Tag 地址。CPU 发出的 8 位地址，首先通过其中的 2 位索引地址找到对应的组，每个组内有两个 Cache 行，地址中的 3 位 Tag 地址将被送往这两个 Cache 行的 Tag 存储器进行比较，如果行内容有效，且其中一个行的 Tag 与当前 Tag 地址相等，则表示 Cache 命中，并通过命中的行号和第一级 Mux 选择输出 Cache 行内容，并通过二级 Mux 和行内偏移选择相应的字节。

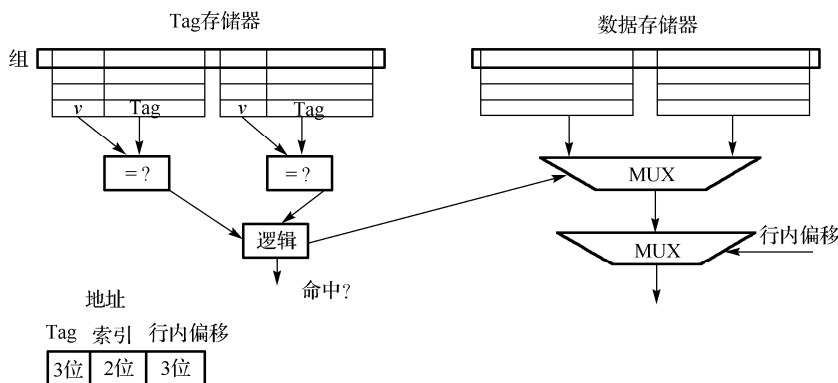


图 5-6 8 位地址 2 路组关联 Cache

图 5-7 所示是 4 路组关联映射，其基本原理与 2 路组关联映射基本相同。

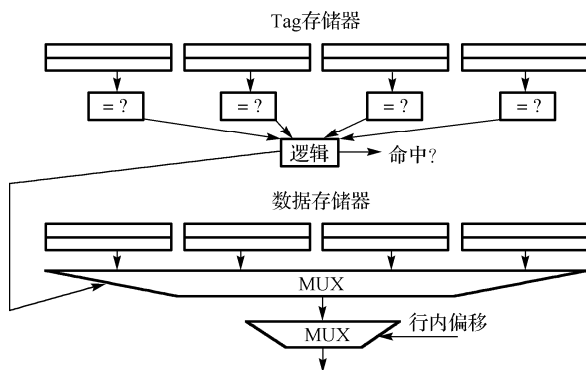


图 5-7 8 位地址 4 路组关联

一般来说，在 Cache 行大小和容量相同的前提下，组关联数越高（也就是一组内的行数 k 越大），命中率也越高，但由于每次访问都需要同时比较 k 行的 Tag 地址，而且多数设计在比较 Tag 的同时将读取所有的组内行（为了减少 Cache 的访问延时），由 Tag 比较的结果选择相应命中行的内容输出，这些都造成了单次访问 Cache 的能耗增高。另一方面，由于对于 Tag 的比较是同时进行的，组内的行数越多，比较器的设计也越复杂，比较器的延时也可能更大，从而造成访问延迟变大（也就是时钟频率降低）；而组关联较低时，更容易引起 Cache 的冲突，而 Cache 冲突势必带来大量额外的访外存操作，从而造成 Cache 性能的急剧下降与系统能耗的上升。但正如前面所分析的，组关联数的提高是有代价

的，不管是在功耗上还是在访问延迟以及芯片面积上，因此实际设计中往往需要根据应用的需求进行折中设计。

Cache 容量也称为 Cache 大小，相当于 Cache 行数乘上每行可容纳的字节数。一般而言，随着 Cache 容量的增加，其不命中率将逐渐下降；但容量的增加意味着芯片面积和能耗的增加，同时也意味着访问延迟的增大。因此，当命中率达到一定高度时，通过增加面积得到的命中率收益递减而引入的能耗与增加的芯片面积却显著提升。

4. Cache 的行大小

Cache 的基本组织单位是行，每个 Cache 行的容量为几字节到几十字节之间不等。若 Cache 行设置过大，不命中时需要换入换出的数据量就将增大，从而增加了 Cache 填充的延时；由于程序的时间局部性原理，每个被填充的 Cache 行仅有一部分内容会被经常访问到；若 Cache 行选择过小又不能更好地利用程序局部性的特征。因此一般而言，每个 Cache 行的容量为 16~64 字节时，对大多数基准测试程序都有较好的优化效果。

5. Cache 的替换策略

由于 Cache 的行数远远小于主存的行数，因此在程序的执行过程中会不断有新行填充到 Cache 行，这时原来存储的那个行必须被替换掉。对于直接映射，一个确定的存储行只对应一个 Cache 行，因此直接替换存在 Cache 的行就可以了；而对于组关联映射和全关联映射，至少每个组中存在两个 Cache 行（对于 2 路组关联而言），因此需要一种替换算法来选择被替换的行。为了提高访问速度，通常替换算法由硬件完成。一般有 4 基本的替换策略。

（1）随机替换。

用伪随机数来产生要替换的行。这种方法中替换的块具有较大的随机性，但在实际测试程序下，随机替换策略可以有相当不错的表现。随机替换策略的另一个好处是硬件实现比较简单。

（2）最近最少使用（LRU）替换策略。

该替换策略选择那些最近最少使用的行，也就是那些未被引用过且在 Cache 中驻留时间最长的行。但是该策略需要 Cache 记录每个行的访问次数，利用过去的信息来预测将来的使用情况。也正因为如此，LRU 替换策略虽然表现不错，但是由于需要为每个 Cache 行设计相应的多个比特的计数器，并且在替换发生时还需要选择最少使用的行，因此其硬件设计较为复杂。在实际 Cache 设计中很少采用纯粹的 LRU 替换策略，而是采用近似的 LRU（PLRU）替换策略。

（3）伪 LRU 替换策略。

由于 LRU 算法的实现复杂性，实际 Cache 设计中通常采用该算法。在该算法中，每个 Cache 行设置一位控制位，初始状态下组内的所有行都为“0”，如果访问该行，则翻转该位为“1”。替换发生时，总是选择状态为“0”的行，如果有多个“0”，则随机选择其中之一进行替换。如果所有行都为“1”，则除了最近访问的行外，其他所有行翻转为“0”。很多研究表明，PLRU 算法在性能上与 LRU 算法接近，但其硬件实现却要简单得多。

（4）Round-Robin 替换策略。

该替换策略轮流替换组内的 Cache 行，比如对于 4 路组关联的 Cache，首先替换组内的第一行，然后是第二行，直到第四行，然后再回到第一行，以此类推。该替换策略的硬件实现也比较简单。

6. Cache 的写策略

通常情况下 CPU 对指令的访问是只读的，因此在发生替换时，Cache 只需要将新装载的行直接覆盖被替换的行即可。但对于数据的访问，Cache 必须考虑被替换出去的 Cache 行是否发生过改变。如

果 Cache 行中的数据已被修改,则 Cache 还需要将修改后的内容写回到片外存储器。在 CPU 执行写操作时,常用的写策略(Write Strategies)如下。

(1) 写穿策略(Write Through)。

如果写命中(所谓写命中是指 CPU 写操作所访问的地址段正好被缓存在 Cache 行中),则将写数据写入 Cache 的同时写入主存;写不命中(也称为写缺失)时只将写数据写入主存。在写操作时微处理器必须等待缓慢的主存储器操作(通常较 CPU 速度低 1~2 个数量级)。

(2) 带 Write Buffer 的写穿策略。

如果写命中,则将写数据写入 Cache 相应单元的同时写入 Write Buffer(写缓冲);写不命中时只将写数据写入 Write Buffer。所谓写缓冲实际上是一块与 CPU 速度相同的片上存储器, CPU 发起的写操作可以立即保存到写缓冲中,不用等待慢速的片外存储器。写缓冲在此后存储器空闲的时间里将数据写入主存。由于 Write Buffer 中可能含有有效的数据,因此 CPU 在每次执行读操作时,控制逻辑都会先判断所要读取的数据是否在 Write Buffer 中。

(3) 写回策略(Write Back)。

写回策略是指 CPU 在执行写操作时,如果写命中,被写的数据不再写入主存而只写入 Cache。仅当 Cache 更新并把脏块替换出去时(脏块是指数据被更新过,Cache 中的数据与内存数据不一致的 Cache 块),才把脏块写回到主存中。而如果写不命中,写数据是直接写入主存还是将主存的块重填入 Cache 后写入 Cache 由分配策略决定。写回策略的 Cache 一般都分配给每个 Cache 行一个 dirty 位,当一个 Cache 行中的任何一个字被修改时,该行 Cache 的 dirty 位被置 1,否则该行 Cache 的 dirty 位保持为 0。当某一个 Cache 行需要被替换出去时,如果对应 Cache 行的 dirty 位为 1,则需要把该行 Cache 的内容写回到主存后,新的主存块才能被重填进来;如果对应 Cache 行的 dirty 位为 0,则直接把主存中的块填充入该行 Cache 即可。

我们主要从下面几个方面来比较写回策略和写穿策略的优缺点。

(1) 可靠性。

写回策略的可靠性比写穿策略要差。因为写穿策略能保证 Cache 始终是主存的正确副本。如果 Cache 发生了错误,可以从主存中纠正。

(2) 与主存的通信量。

一般来说,写穿策略与主存的通信量要高于写回策略。这个问题可以从两个方面来理解:一方面,由于 Cache 具有很高的命中率,对于写回策略, CPU 的大多数写操作不需要写主存,只需写 Cache。而写穿策略对于每次写操作都要写到主存中;另一方面,对于写穿策略,当 CPU 发出的写操作在 Cache 中没有命中时, CPU 每次只写一个字到主存而对于写回策略,当 Cache 发生读未命中时,可能会由于 dirty 位为 1 而要将 Cache 的一个行写回到主存。总的来说,写穿策略增加了写操作的开销,因为它在每次写 Cache 时,同时也会写主存。而写回策略是在 Cache 缺失时把写主存的开销集中起来,一次性地写一个 Cache 行到主存里。

(3) 控制的复杂性。

写回策略比写穿策略的控制逻辑复杂。写回策略需要 dirty 位,同时,写回策略的纠错技术比较复杂。

(4) 硬件实现的代价。

一般来说,写穿策略的硬件实现代价比写回策略要小,因为写穿策略不需要 dirty 位,但由于写穿策略通常会采用一个 Write Buffer,因此写穿策略的硬件实现代价就比写回策略要略大。

写穿策略与内存通信量较大,而程序执行过程中又需要考虑数据的一致性,因此写回策略得以出现并得到发展。

7. Cache 的分配策略

当进行数据写操作时，会发生 Cache 未命中的情况，这时可以有两种分配策略：读操作分配策略（Read-allocate）和写操作分配策略（Write-allocate）。

对于读操作分配策略，在指令或数据读缺失时为主存块分配 Cache 行，将主存块读入 Cache 行后，再把目标数据返回给 CPU。在数据写缺失时并不分配 Cache 行给主存块，只把数据写入主存。而对于写操作分配策略，无论 Cache 是读缺失还是写缺失，都会分配相应的 Cache 行给主存块，在主存块填入 Cache 行后，再对 Cache 进行读或写操作。

由于写操作分配策略提高了 Cache 的命中率，但同时增加了 Cache 对主存进行预取的次数，因此这种策略对于整体系统性能的影响与程序中读操作和写操作的数量有关。

5.2.3* Cache 缺失与访问冲突

1. Cache 缺失（不命中）的分类

Cache 不命中有多种分类方式，其中根据 Cache 访问类型进行分类，可以分为指令 Cache 读不命中、数据 Cache 读不命中以及数据 Cache 写不命中 3 类；若根据 Cache 不命中的成因进行分类，则又可以分为：冷不命中（Cold Miss）、容量不命中（Capacity Miss）及冲突不命中（Conflict Miss）。首先给出 Cache 冲突的一般定义：由于传统 Cache 受限于组关联数，从而导致尚有价值的 Cache 行被过早换出（Evict）。事实上，由 Cache 冲突带来的缺失大多是可以避免的。全关联的 Cache 可以在 Cache 中的任意位置（必须以 Cache 行大小对齐）为主存数据分配新的 Cache 行，因此不存在 Cache 冲突不命中，而只存在容量不命中（也就是 Cache 中的所有行都被占用了，对新行的访问必然需要替换出一个旧行），但由于其能耗太大，占用芯片面积也太大，因此在实际嵌入式存储子系统设计中很少用到。绝大多数 Cache 使用有限组关联或直接关联，因此在映射主存数据时会受到 Cache 组关联数限制，从而导致 Cache 冲突不命中。

冷不命中也被称为强制性不命中（Compulsory Miss），通常是由于 CPU 所访问的指令或者数据还没有被装载到对应的 Cache 行中所引起的访问不命中。影响强制性不命中出现次数的因素是 Cache 行的容量与是否配置指令/数据预取缓冲。事实上，这两种做法的目的都是增加 Cache 的预取字节数，Cache 一次预取的字节数越多则意味着越多的连续地址空间避免了冷不命中。然而，通过增加 Cache 的预取能力以降低其冷不命中的绝对数目/冷不命中率的做法本身有其弊端：当程序流水线被打断，或者出现 Cache 由于其他两种缺失而被迫更新 Cache 内容时，增加的预取字节将提升每次换入换出的成本。这其实也就是降低了总的命中次数/命中率，但牺牲了单次不命中的惩罚。

容量不命中是因为 Cache 体中已经被填满了其他的数据或指令，为了装载新的数据或指令必须将现有的某个 Cache 行的内容替换出去，才能装载新的数据；而冲突不命中是因为 Cache 的地址映射机制造成的。容量不命中和冲突不命中比较难以区分，图 5-8(a)~(d)分别给出了基准测试程序 JPEG-enc 在配置 4KB-直接关联、8KB-直接关联、8KB-2 路组关联以及 8KB-4 路组关联的数据 Cache 的读缺失分布。

图 5-8 中 X 轴向表示出现缺失的虚存地址空间，以 512B 页大小为单位，Y 轴表示整个程序执行过程中某虚存页上冲突的总次数^①，Z 轴表示不同的时隙（20slot, 40slot, …）。由图 5-8(a)和(b)的对比可以发现，对于基准测试程序 JPEG-enc 而言，数据 Cache 的容量所带来的缺失与冲突带来的缺失在分布上是有所不同的，随着容量的增加，由 4KB 增大到 8KB，而组关联数不变，则总的缺失数目以及每

① 注意，4 个子图的 Y 轴坐标是不一致的，图 5.8(a)的最大值是 300，图 5.8(b)和图 5.8(c)是 150，而图 5.8(d)是 80。

段地址空间上的绝对缺失数目都有所降低，但是出现冲突的地址空间基本没有变化，而且不同地址空间上出现缺失的数目的比例基本没有变化。由此可以分离出 4KB 和 8KB 在相同组关联数的前提下的容量缺失。相似地，由图 5-8(b)、(c)和(d)的对比，我们可以看出在 Cache 容量一定的前提下，Cache 缺失随着关联度的变化，其分布具有如下特征。

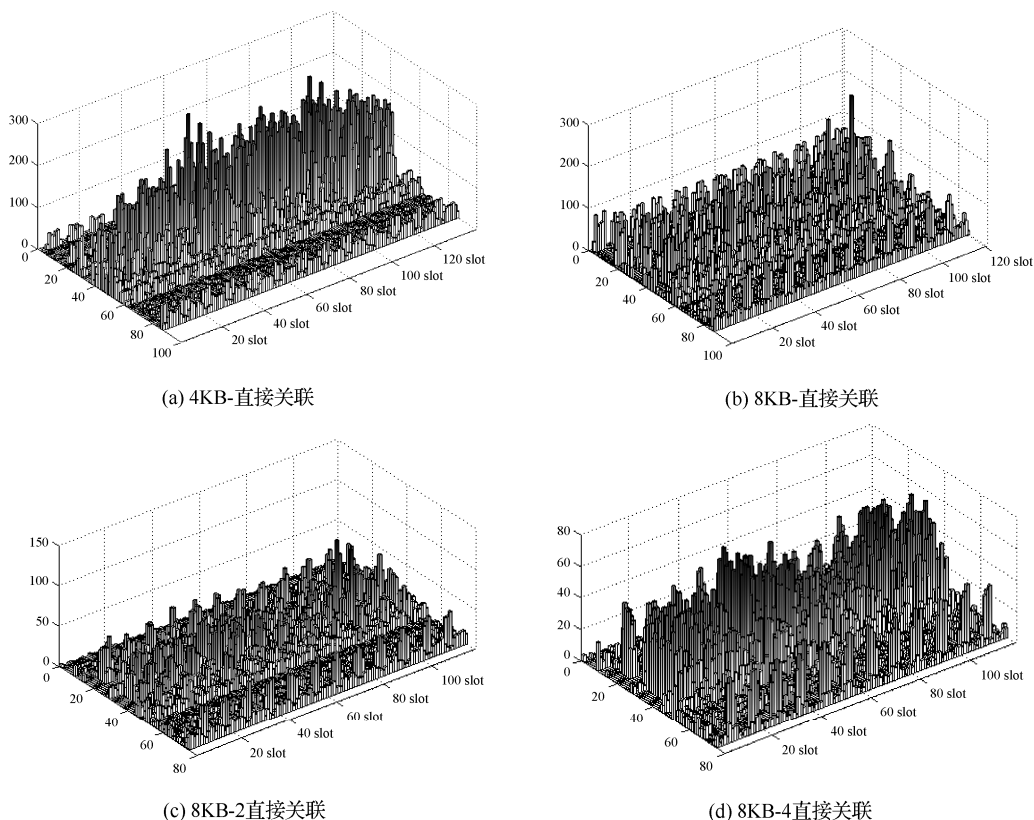


图 5-8 基准测试程序 JPEG-enc 在不同 Cache 配置下数据 Cache 缺失分布

(1) 随着组关联数的提升，绝对数据 Cache 缺失数目发生了较大幅度的下降，这是由于关联度提升后多个工作集的数据可以映射到 Cache 行的数目倍增，纯粹由于每路 Cache 容量不足带来的 Cache 换入换出的数目减少。但必须指出这是在牺牲一定的芯片面积（增加了比较电路）、增加了单次访问 Cache 的能耗以及降低了单次访问 Cache 的速度（也是由于增加的比较电路带来的延时）的前提下达到的。此外，单纯增加 Cache 的组关联数并不能最小化 Cache 缺失。事实上，如图 5-9 所示，A、B 两个区域各自分别负责映射经常访问的 3 块和 2 块外存地址空间，它们的确可以从提升 Cache 组关联数获得收益，然而原本并没有由于 Cache 组关联度较低导致 Cache 缺失的 C 区域也由于提升整体 Cache 组关联度而增大了不必要的单次访问能耗与芯片面积，这就是整体提升 Cache 组关联数带来的弊病。同样的情况也发生在整体提升 Cache 容量上，提升 Cache 容量的确可以减少一部分容量性不命中，然而由于程序运行具有较为明显的时间局部性，提升的 Cache 容量并不能加速访问所有的地址空间，却必须付出 1 倍、2 倍甚至多倍的 Cache 面积。

(2) 同样，随着组关联数的提升，某些地址空间上曾经集中的缺失的确被大量削减，但是被削减的部分并没有累加到其他地址空间。即加大整个 Cache 的关联性事实上仅对地址空间的某些部分的冲突有缓解作用，而对其他部分效果并不明显。通过实验可以发现：在整个程序执行过程中真正引起

Cache 行冲突的地址并非简单的均匀分布，而如果由于某些特定地址空间出现大量冲突性不命中就增大整个 Cache 的关联性是得不偿失的。国际上最近出现了所谓的动态重构的 Cache 研究，其基本思路之一就是在程序执行过程中将 Cache 的一部分在需要时配置为一定关联度，从而满足程序对 Cache 的要求。然而这种办法本身大大增加了 Cache 的复杂性，同时由于嵌入式系统对实时性有着更为严格的要求，而 Cache 本身的动态配置必然引入额外的配置时间，因此其实时性相较普通 Cache 更为糟糕。

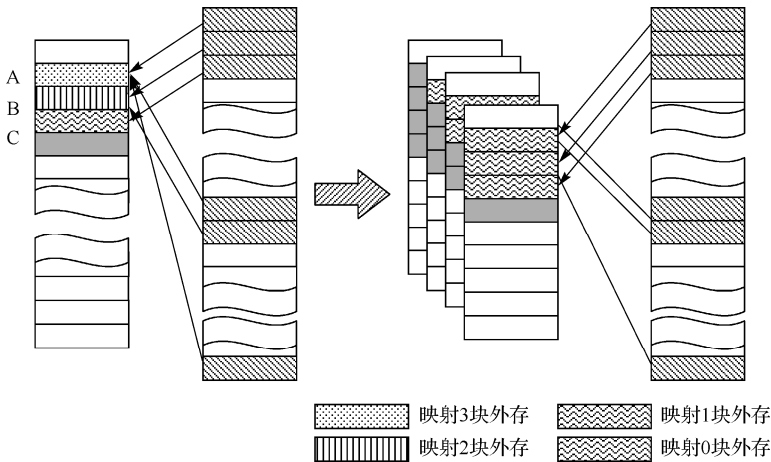


图 5-9 增加 Cache 组关联数对 Cache 冲突的影响

为了减少基准测试程序的某些地址空间的 Cache 冲突带来的缺失数目，直接采用关联性较大的 Cache 不仅会直接增大芯片面积（比较电路的面积），而且对于能耗来说也并非最佳的选择，理由如下：

- （1）由学界广泛采用的标准 Cache 模型——CACTI（见扩展阅读[2]）的仿真结果可知：Cache 的能耗中很大一部分（25%左右）是来自比较电路和 Tag 存储器，而对于数据段（包括堆栈数据）空间局部性较好的基准测试程序而言，除去需要较高关联度来消除 Cache 冲突不命中的地址空间以外，其他大量地址空间所映射的 Cache 实际上并不需要额外的组关联数，从而造成大量能耗被消耗在不需要的地方。
- （2）由于程序在不同执行阶段对数据的要求是不同的，因此即使不考虑空间上局部性带来的组关联使用率较低的问题，也还需考虑时间局部性带来的组关联使用率较低的问题。

2. Cache 性能的优化方法

处理器与存储器之间的速度差距变得越来越大，访存速度成为约束处理器性能进一步提高的瓶颈。Cache 作为处理器与主存之间的高速缓冲，填补了处理器与主存之间巨大的速度差距，因此如何提高 Cache 的性能成为人们研究的重点之一。

由于平均单次存储器访问时间=命中率×命中时间+缺失率×缺失代价（其中命中率+缺失率 = 100%），所以为了提高 Cache 的性能，可以从降低 Cache 缺失率、降低 Cache 缺失代价以及减少 Cache 命中时间等几个方面入手。

正如前面所介绍的，按照 Cache 产生缺失的原因不同，可将 Cache 缺失分为 3 类：强制缺失（也称为冷缺失）、容量缺失和冲突缺失。根据 Cache 发生缺失时的种类和原因，下面介绍几种降低缺失率的方法。

- （1）增加 Cache 行大小：这种方法是降低 Cache 缺失率的最简单的方法。一般来讲，在 Cache 容量确定的情况下，缺失率刚开始随着 Cache 行的增大而降低，后来则随着 Cache 行的增大而上升；一方面增加 Cache 行大小提高了对空间局部性的利用，导致缺失率下降；另一方面，由于 Cache 容量一

定, 增大 Cache 的行大小意味着总的 Cache 行数减少, 导致冲突失效增加。这两者相互作用, 所以导致 Cache 缺失率先降低, 后升高。另外, 随着 Cache 行的增大, 其缺失代价也会进一步增加 (因为重新填充 Cache 行的时间增加了)。

(2) 提高关联度: 提高 Cache 的关联度会导致从下一级存储器调入 Cache 行时可以选择的 Cache 行更多, 这样能够降低冲突失效, 从而降低失效率。但是, 关联度的增加会使得 Cache 的命中时间增加 (因为 Tag 比较器多了, 硬件延迟增加), 过度增加关联度并不能够一直降低访存时间。同时, 随着关联度的提高, Cache 的功耗也会进一步增大。

(3) 硬件预取技术: 预取技术指在处理器访问指令或者数据之前将其进行预取, 通常是直接放入 Cache 或者其他外部的缓冲器。硬件预取技术提高 Cache 性能指通过扩展 Cache 管理子系统在指令或者数据发生 Cache 缺失的时候, 不但将被请求的块调入 Cache, 还将被请求块的下一个块也预取过来。

(4) 软件预取技术: 软件预取技术和硬件预取技术相似, 只是在进行预取时并不一定预取被请求块的下一个块, 而是通过软件指定被预取的块。软件预取一般通过在程序中增加预取指令来实现, 通常由编译器或者专门的程序变换工具来实现。由于软件预取能够根据对程序进行分析来指定预取内容, 可以减少不必要的预取, 从而获得好的效果。

(5) 代码变换: 利用 Cache 层次来降低处理器和存储器之间的速度差距的影响, 主要是依据程序的局部性原理, 提高程序自身的局部性能有效地提高 Cache 性能。改善程序的局部性要么改善程序指令的局部性, 要么改善程序数据的局部性, 其中后者更占主导地位。在改善数据局部性中, 代码变换通过改变指令的执行顺序, 不但优化数据的时间局部性, 还优化数据的空间局部性。

(6) 数据变换: 在优化程序的局部性中, 除了通过代码变换来改变程序指令的执行顺序, 还可以通过数据变换来改变程序数据的布局, 充分提高程序数据的空间局部性, 从而提高数据 Cache 的性能。

为了降低 Cache 缺失代价, 通常有读缺失优先、子块搁置、关键字优先、非阻塞 Cache 以及采用多级 Cache 等技术。

(1) 读缺失优先于写回的方法是当发生读缺失时, 将脏块复制到一个缓冲区之后就去读下层存储器, 而不是先把脏块写回到下层存储器之后再继续进行读操作, 这样使得读操作不需要等待, 可以加快读操作的速度。处理器在获得读操作取回的数据后可以继续执行, 从而减少了处理器停顿的时间。

(2) 子块搁置方法 (sub-block placement) 的主要思想是将大容量 Cache 块划分为若干子块, 每一个子块都设立一个有效位, 当缺失时只需要将单个子块读入 Cache, 这样既可以通过增大 Cache 行容量来减少缺失率, 同时又可以降低缺失代价。

(3) 关键字优先的方法是在读缺失时首先向下层存储器请求缺失的字, 这样当缺失的字返回后处理器就可以继续执行, 而不需要等到所有的 Cache 块数据都返回, 减少了处理器停顿的时间, 降低了缺失代价。

(4) 非阻塞 Cache 技术是通过数据 Cache 在缺失时允许后续访存操作继续执行来实现的。当一个 Load/Store 操作在 Cache 中不命中时, 一方面向下层存储器发出缺失请求, 另一方面并不阻塞后续的访存操作, 允许后续操作继续访存, 在缺失情况下继续提供 Cache 命中的功能, 从而降低了缺失代价。

(5) 在一级 Cache 和存储器之间提供一个容量更大的二级 Cache, 则可以捕捉在一级 Cache 不命中时对存储器的大多数访问, 因而减少了缺失代价。一些处理器甚至提供了三级 Cache 来进一步降低缺失代价。

Cache 命中时间的重要性在于它能够影响到处理器的时钟频率。减少 Cache 命中时间不但能够有效地减小了整个访存时间，而且能够通过增加时钟频率提高系统许多其他方面的性能。常用的减小 Cache 命中时间的方法有两种。

（1）容量小、结构简单的 Cache。

在组关联的 Cache 中，地址被分为 3 段：Tag 地址、索引地址和行内偏移地址。Cache 通过索引地址选择相应的组，并将 Tag 地址同时送往该组内的所有行（两路组关联就是 2 行，四路组关联就是 4 行，以此类推），同时比较所有行的 Tag 地址是否与该访问地址的 Tag 相等。在进行 Tag 比较的同时，被选中组的所有行都会被读出并送往一个 Mux，由 Tag 比较的结果选择输出哪个行的内容。从这个过程中我们知道，Tag 比较和行读出（假设本次访问是读操作）是同时进行的，其中较慢的那个过程决定了命中时间。组内的行越多，比较时间就越长。另一方面，Cache 体越大，读出时间也越长。总之，结构越复杂，容量越大都会造成访问时间变大。因此，结构简单的 Cache，如直接映像 Cache，或者组关联数较小的小容量 Cache 都可以有效地减少命中时间。在一般系统中，第一级 Cache 都选用容量小、结构简单的方案，因为一级 Cache 对于访问延时更敏感。

（2）采用虚拟地址。

在采用虚拟存储器的系统中，每次访存都需要将虚拟地址转换为物理地址。在 Cache 中采用虚拟地址，检测 Cache 是否命中时直接采用虚拟地址，无需进行地址转换。在虚地址发给 Cache 的同时，MMU 完成虚地址与物理地址的转换。如果命中，则减少了 Cache 的访问时间；如果不命中，则通过物理地址转载新的 Cache 行。通常情况下，在现代高性能嵌入式为处理器中一级 Cache 采用虚地址，而二级 Cache 采用实地址。虚地址 Cache 的缺点是功耗可能较实地址 Cache 要高。

5.2.4* Cache 一致性问题

多核多处理器意味着在单芯片上有多个处理器（CPU），这些处理器可能会共享一个公共的物理地址空间。Cache 共享数据带来了一个新的问题，由于两个不同的处理器所保存的存储器视图是通过各自的 Cache 得到的，如果没有其他的防范措施，两个处理器可能分别得到两个不同的值。

一般情况下，如果在一个存储器系统中读取任何一个数据项的返回结果总是最近写入的值，那么可以认为该存储器具有一致性。这个定义尽管看起来是正确的，但仍很模糊而且过于简单，实际情况要复杂得多。这个简单的定义包括了存储器系统行为的两个不同方面，它们对于编写正确的共享存储程序是至关重要的。第一个方面称为一致性（Coherence），它定义了读操作可以返回什么样的值；第二个方面称为连贯性（Consistency），它定义了写入的数据什么时候才能被读操作返回。

如果一个存储系统满足如下条件，那么认为该存储系统是一致的。

- 处理器 P 对位置 X 的写操作后面紧跟着处理器 P 对 X 的读操作，并且在这次读操作和写操作之间没有其他处理器对 X 进行写操作，这时读操作总是返回 P 写入的数值。
- 在其他处理器对 X 的写操作后，处理器 P 对 X 执行读操作，这两个操作之间有足够的时间间隔并且没有其他处理器对 X 进行写操作，这时读操作返回的是其他处理器写入的数值。
- 对同一个地址的写操作是串行执行的。也就是说，任何两个处理器对同一个地址的两个写操作在所有处理器看来都有相同的顺序。

第一个性质保证了程序的顺序，即使在单处理器中也要保证这个性质（还记得乱序处理器的写后读问题吗？）。第二个性质定义了存储器的一致性意味着处理器 P 总是应该读到最新写入的数值（不管这个写入操作是由处理器 P 自己发起的，还是由其他处理器发起的）。

写操作串行化的要求更加细致，但也同等重要。假如我们没有将写操作串行化，处理器 P1 写入地址 X 之后，紧跟着处理器 P2 也写入地址 X。写操作串行化保证了每个处理器都能在某个时间看到

P2 写入的结果。如果没有将写操作串行化,就会出现一些处理器先看到 P2 写入的结果再看到 P1 写入的结果,从而可能保留了 P1 写入的数值(此时 P1 写入的值其实已经是无效的了)。避免这种情况最简单的方法就是保证对同一个地址的写操作在所有处理器看来具有相同的顺序,这个性质称为写串行化(write serialization)。

1. 实现一致性的基本方案

在支持 Cache 一致性的多处理器系统中,Cache 提供共享数据的迁移和复制。

- 迁移:数据项可以移入本地 Cache 并以透明的方式使用。迁移不但减少了访问远程共享数据项的延迟,而且减少了对共享存储器带宽的需求。
- 复制:当共享数据被同时读取时,Cache 在本地对数据项做了备份。复制减少了访问延迟和读取共享数据时的竞争现象。

对这种迁移和复制的支持对于访问共享数据的性能来说是至关重要的,因此许多的多处理器引入硬件协议来维护 Cache 一致性。这个用于维护多个处理器一致性的协议称为 Cache 一致性协议(Cache coherence protocol)。实现 Cache 一致性协议的关键在于跟踪所有共享数据块的状态。

最常用的 Cache 一致性协议是监听(snooping)协议。每个含有物理存储器中数据块副本的 Cache 还要保留该数据块共享状态的副本,但是并不集中地保存状态。Cache 可以通过一些广播媒介(总线或者网络)访问,所有的 Cache 控制器对媒介进行监视或者监听,来确定它们是否含有总线上请求的数据块副本。

下面将介绍用共享总线实现基于监听的 Cache 一致性方法,任何可以向所有处理器广播 Cache 缺失的通信媒介都可以用来实现基于监听的一致性机制。这种向所有 Cache 广播的方法使得监听协议的实现变得简单,但是也限制了其扩展性。

2. 监听一致性协议

监听协议使用共享总线连接多个处理器内核的私有 Cache 和主存,共享总线保证所有处理器内核的数据请求串行执行。任何处理器发出的数据请求将被广播到所有处理器内核节点,所有处理器内核的 Cache 控制器都时刻监视着总线。如果收到读请求,数据所有者将把有效数据返回给发出此请求的处理器内核;如果收到写请求,拥有有效副本的处理器内核便无效(Invalidate)或更新本处理器上的数据,数据所有者将把有效数据返回到发出此请求的处理器内核。数据所有者可能是处理器内核,也可能是主存。

下面介绍最基本的 MSI 总线监听协议。由于 Cache 本身的特点,对 Cache 的访问是以 Cache 行为单位进行的,一个 Cache 行通常为 32 字节或 64 字节,统计结果表明这样大小的 Cache 行能使 Cache 的命中率达到最高。MESI 协议是采用写回和写无效策略的监听协议。MESI 协议以系统中 Cache 行的状态定义的首字母组合而成,所涉及的状态如下。

- M: Modified, 表示该行数据有效,且数据被修改了,和内存(下一级存储器)中的数据不一致,数据只存在于本 Cache 中。对该行进行读写访问均不会改变 Cache 行的状态,当该行需要被替换出去时,必须将其写回到主存中,因为它是系统中唯一一个正确的副本。
- E: Exclusive, 表示该行数据有效,数据和内存中的数据一致,数据只存在于本 Cache 中(也就是其他处理器的 Cache 中没有缓存相同的内容)。对该行的读操作不会改变该 Cache 行的状态,写操作会将该行的状态变为 M,以此表明该数据是系统中唯一正确的副本。
- S: Shared, 表示这个 Cache 行中的数据未被修改且至少有另一个 Cache 包含了此数据块的副本。当包含此数据块的 Cache 行被替换时,无需更新主存中对应的副本。读操作不会改变该行

的状态。但是写操作不仅需要将该 Cache 行状态改为 M，还要向总线广播一个无效信号通知其他 Cache 使相应的 Cache 行无效。

- **I: Invalid**，表示该 Cache 行数据处于无效状态，如果此时处理器需要读此块数据，则必须从主存或者其他含有有效数据的 Cache 中读取此数据块。对该行读操作时，需要向总线广播一个数据请求信号，等待其他 Cache 监听到该请求信号，并让拥有此 Cache 行的正确副本的 Cache 将其发送回来。同时，发送正确副本的 Cache 对应的行状态变为 S，收到正确副本的 Cache 在读操作完成后将数据状态改为 S。

MESI 监听协议是比较经典的 Cache 一致性协议。MESI 协议最初由 Illinois 大学的研究者提出，所以又称 Illinois 协议，自推出后得到了广泛的应用。目前主流的处理器，如 x86、Power、MIPS 和 ARM 处理器，均使用类 MESI 协议维护 Cache 一致性。

3. 目录一致性协议

除了分布地保存共享块状态的监听 Cache 一致性协议，基于目录的 Cache 一致性协议将物理存储器的共享块的状态存放在一个地点，称之为目录（directory）。尽管基于目录的一致性比监听式一致性的实现开销略高一些，但是这种方法可以减少 Cache 之间的通信，因此可以扩展更多的处理器。

在目录协议中，最重要的功能部件是目录。目录用于存储存储器中每一个共享数据块的状态信息和其他相关信息。目录中存有很多目录项，一个特定目录项记录了此共享数据块的状态信息和所有包含此共享数据块的远程副本位置指针。

目录协议的工作原理为：通过为共享数据设置目录项以记录其信息，从而在处理器核请求到来之后做出恰当的响应。当处理器核需要访问一个数据块时，需要首先向目录发出请求，在查找目录完成后，由目录将请求转发给数据块所有者，最后由数据块所有者向源请求处理器核发送其请求的数据块，以上操作完成后，将源请求处理器核添加为此数据块的一个共享者。

在多核环境下，一个共享数据块在一个特定时间段内可能被多个处理器核访问。可能出现多个处理器核并发地对同一个共享数据块发出写请求，这时目录提供了对请求排序的功能，使得多个处理器核对一个数据块的请求以队列的形式顺序执行，目录在处理队列中一个写数据请求时，才去触发另一个针对此块的数据请求继续执行。由于对同一数据块的读请求不会修改数据块的值，允许对多个读请求并行处理。

4. 多核架构 Cache 存储层次下的数据一致性

在典型的多核处理器架构中，每个单个处理器核包含一个私有的 L1 Cache（一般包括指令 Cache 和数据 Cache），所有处理器核共享一个 L2 Cache。这样的设计保证了共享二级 Cache 可以有效满足片内其他处理器核的读数据请求和远程处理器的一致性事务请求。同时对于写请求的处理不单独进行，而是通过共享二级 Cache 的存储队列合并集中。二级 Cache 设有多个写回缓冲、监听队列和 MSHR（Miss-Status Handling Registers）表项。写回缓冲、监听队列表项用于处理已经抵达的一致性事务，MSHR 表项用于记录未完成的事务。为对二级 Cache 中的数据副本进行有效管理，在二级 Cache 的标志存储器中设置一个共享变量，记录二级 Cache 中任何一个 Cache 数据副本在两个处理器核中的包含情况。在有一个本地处理器核或远程处理器对一个共享数据副本发出写请求时，就可以通过上述的共享变量找到拥有此共享数据副本的本地处理器核，并向其发送写无效信息，以保证写操作下 Cache 数据的一致性。

5.2.5 Cache 和 SPM 的比较

一般而言，SoC 存储子系统可以划分为片上和片外两个部分。片上存储器通常由高速 SRAM 构成，

包括 Cache 和 SPM。SPM (Scratch-Pad Memory) 是集成在 SoC 芯片内部的一块高速 SRAM。与 Cache 完全由硬件控制不同, SPM 通常由软件控制。由于 SPM 的硬件要比 Cache 简单得多且由软件管理, 因此 SPM 更加灵活且能耗较同样容量的 Cache 要低。

从图 5-10 中可以清楚地看出, Cache 位于主存和处理器内核之间, 不占用独立的地址空间, 保存最近一段时间内处理器访问到的主存内容。在需要进行数据指令读取操作时, 处理器总是从 Cache 中读取, 根据地址检查是否命中。如果命中, 则直接将数据或指令传送给处理器; 否则就从主存储器中该地址附近读取固定数目的字并送入 Cache, 然后再送给处理器。从访问的局部性原理来考虑, 当把一块数据送入 Cache 以满足某一次存储器访问时, 将来访问该块中其他字的可能性也是极大的。这样就在减少外部慢速存储器访问的同时也降低了功耗。Cache 经常与写缓冲器 (write buffer) 一起配合使用。写缓冲器是一个非常小的先进先出 (FIFO) 存储器, 位于 Cache 与主存之间。它的目的就是 will 处理器内核和 Cache 从较慢的主存写操作中解脱出来。

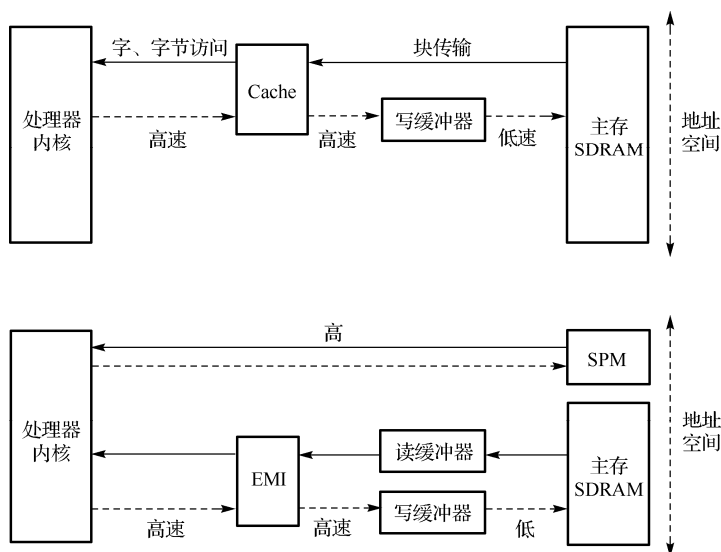


图 5-10 Cache 和 SPM 在系统中的位置

SPM 通过片上高速总线和处理器直接连接, 独占一段地址空间, 保存着部分指令和数据。当处理器需要进行存储器访问时, 总线译码器首先根据访问地址 (此时的地址已经经过 MMU 转换为物理地址) 判断地址空间, 选择 SPM 或 SDRAM (其实是 SDRAM 的控制器)。前者可以直接操作, 后者则需要通过外部存储器接口模块来访问, 并且需要时序上的等待和片外总线驱动等, 从而极大地影响了速度并增加了功耗。

Cache 由硬件管理, 大部分情况下对于软件透明, 能自动将频繁访问的指令和数据装载到 Cache。每当 CPU 内核发出数据请求时, Cache 的硬件首先检查该数据是否已经存放在 Cache 体中。如果数据已经存放在 Cache 中则直接将该数据返回给 CPU (称为 Cache 命中), 通常情况下这只需要一个或几个时钟周期的延时; 如果 CPU 的数据请求不在 Cache 中 (称为 Cache 不命中或 Cache 缺失), Cache 的硬件预取逻辑将自动将被请求的数据从下一级存储器层次搬运到 Cache 体, 一般情况下, Cache 不命中引入的延时将比 Cache 命中慢一个甚至两个数量级。与 Cache 相比, SPM 的结构要简单得多, 通常情况下 SPM 与外部存储器统一编址 (这一点与 Cache 不同)。与 Cache 采用硬件管理不同, SPM 的管理需要通过软件控制决定在什么时间将什么样的代码或数据放入 SPM。虽然 SPM 的管理相对较为复杂, 但是 SPM 依然有超越 Cache 的优点:

（1）SPM 的访问时间是固定的（通常也是一个或几个时钟周期），这对于强调实时性的系统具有很大的吸引力。

（2）特定容量的 SPM 比同样容量的 Cache 占用更少的芯片面积。这是因为 SPM 没有 Tag 存储器和地址比较逻辑。

（3）同样因为 SPM 的结构简单，SPM 的单次访问能耗要低于 Cache。这主要是因为对 SPM 的访问不需要进行复杂的地址比较。这一点在与组关联的 Cache 相比时更为明显，因为对于组关联的 Cache，每次访问的地址都要并行地送给每个组的 Tag 存储器和数据存储器。

同为片上存储器，Cache 比 SPM 多了一部分比较电路和 Tag Memory，然而 Cache 体部分和 SPM 一样都由片上 SRAM 构成，如图 5-11 所示。从 Cache 的组织结构看，Cache 的设计参数包括容量大小、行大小、关联度、写策略与替换策略（即分配策略）。从存储子系统的结构看，Cache 又包括 L1 和 L2 Cache、数据和指令统一 Cache 和分立式 Cache。Cache 的配置一旦确定下来，对程序员是透明的。

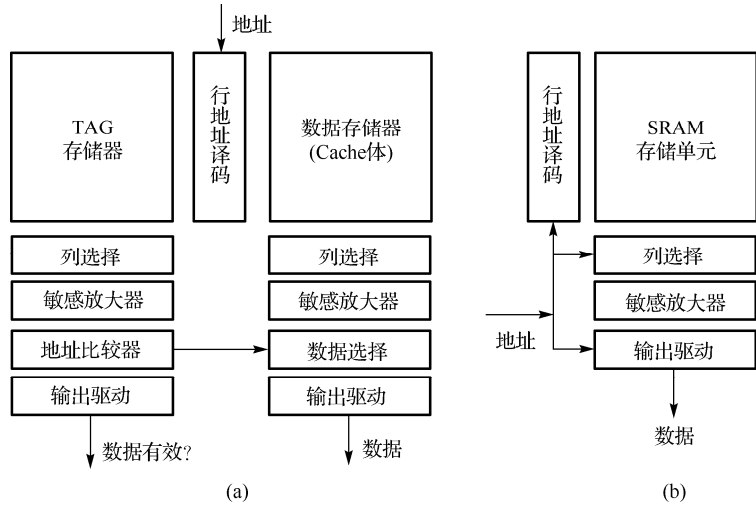


图 5-11 Cache（A）与 SPM（B）的结构比较

SPM 处于处理器可直接访问的地址空间之内，任何对 SPM 地址范围内的访问都会经由总线仲裁器触发。在扣除总线延时的前提下，所有对 SPM 的访问（不论读写）通常都可以在一个或几个时钟周期内完成。由于传统的 SPM 控制器不包含任何辅助管理数据的逻辑电路，SPM 中的所有数据必须由软件显式地管理；由于没有管理逻辑电路带来的额外代价，在相同容量下，SPM 占用的芯片面积和单次访问的能耗都小于 Cache。Cache 由于在发生冲突等情况下访问时间会大幅增加，这使得 Cache 很少应用于对程序执行的确定性要求较高的实时嵌入式系统中。由于 SPM 不是像 Cache 那样由硬件自动完成缓冲数据的换入换出操作，因此 SPM 不存在访问缺失的情况，始终能够保证一个时钟周期的访问时间，从而保证了程序的确定性。总的来说，相较于传统 Cache，SPM 是一种硬件实现更为简单、单次访问功耗更低、占用芯片面积更小而且访问时间可预知的片上存储器实现形式，更适合用于构建实时性要求高的嵌入式系统。Cache 和 SPM 的比较如表 5-1 所示。

表 5-1 Cache 和 SPM 的比较

	Cache	SPM
硬件实现	硬件实现相对复杂，需要附加 Tag Memory 和 Tag Comparor	硬件实现简单，就是基本的 SRAM

续表

	Cache	SPM
地址	从地址上看对程序员透明，自动保存处理器需要访问的指令和数据	从地址上看程序员可见，通过软件制定，借助编译器的优化，实现程序布局
控制方式	硬件控制，自动处理主存和处理器之间代码和数据的传送	通过在程序中内插复制代码完成 SPM 上代码和数据存储
相同容量下功耗	高	低
访问时间	命中时为一个到几个周期 不命中时为十几个到上百个周期	一个或几个周期
相同容量下面积	大	小

根据前文所述，我们可以对 Cache 与 SPM 做以下总结：

- （1）从面积角度看，同 Cache 相比，由于不需要用来存储地址的 Tag 存储器和比较地址的 Tag 比较器，硬件实现更加简单，在相同的制造工艺下，SPM 所占片上面积更小，约为 Cache 的 66%。
- （2）从功耗角度看，SPM 操作功耗更低，相同工艺条件下约为 Cache 的 60%。原因在于，一方面 Cache 的 Tag 比较器和 Tag 存储器等硬件逻辑导致功耗开销，另一方面 Cache 的硬件操作机制常常导致大量随机片外存储器访问并产生大量功耗。相反，由于 SPM 对程序员可见，程序在 SPM 上执行的确定性更高，减少了这种非确定性的片外访问引发的功耗损失。
- （3）从指令执行角度看，对于强实时应用，使用 Cache 无法预测实际最差工作情况，特别是很难保证中断的响应时间。而 SPM 由于程序员可见并受其直接控制，因此行为更加确定。正是由于在面积、功耗、执行时间和实时性等方面的优势，SPM 在各大厂商生产的嵌入式芯片中应用越来越广泛。

5.2.6* ARM Cortex A8 处理器的 Cache 架构

Cortex A8 处理器是 ARM 公司推出的继 ARM11 CPU 后的第一款支持 ARM V7 指令集的高性能 CPU。A8 CPU 支持每时钟发射两条指令，它具有两级 Cache 结构，分别是指令和与数据相分离的一级 Cache 和指令与数据统一的二级 Cache。其中一级缓存可以是 16KB 或者 32KB，采用四路组关联，每 Cache 行 64 字节；二级缓存可以配置为容量 128KB~1MB，采用八路组关联，每 Cache 行 64 字节。Cortex A8 处理器的存储架构如图 5-12 所示。

A8 CPU 的一级缓存采用实地址索引，虚地址 Tag。也就是说，存放在一级缓存中的数据采用虚地址，CPU 发出 32 位的虚拟地址，该地址被同时送往 MMU 和一级缓存，送往 MMU 的地址被分为 18 位虚拟页码和 14 位页内偏移地址（这里以 16KB 页为例，A8 处理器的 MMU 可以配置不同的页面大小，即 4KB、16KB、64KB、1MB 和 16MB）。虚地址的高 19 位作为索引会被送往 TLB 进行检索，以查找与之对应的物理页码地址。TLB 采用 32 个表项的全关联方式进行组织（我们将在下一节介绍虚拟存储器）。在虚地址进行转换的同时，页内偏移地址被分为 7 位一级缓存索引和 6 位行内偏移（每行 64 字节，所以需要 6 位地址），7 位索引将在 128 个 Cache 组中选取映射的组，由于 A8 采用四路组关联，因此每组中有 4 个 Cache 行，虚地址的高 19 位将被同时送往被选中的组中的 4 个地址比较器，以判断是否有存放在该组 4 个 Tag 存储器中的地址与该地址相同，如果有一组 Tag 与出送过来的地址相等，则 Cache 命中，L1 Cache 的内部 Mux 将选择该命中行输出的数据，并送往 CPU。否则，本次访问将造成一次 Cache 缺失，L1 Cache 将发起一次向 L2 Cache 的填充请求，装在新的数据行。

A8 处理器的 L2 级缓存采用物理地址索引，物理地址 Tag。也就是说，送往 L2 Cache 的 Tag 地址是经过 MMU（TLB）转换后的物理地址。L2 级缓存采用 11 位索引，共可以索引 2048 组 Cache 行，由于采用八路组关联，每个组中包含 8 个 Cache 行，每行内部采用 6 个位表示行内偏移地址（每行共

64 字节)。15 位实地址 Tag 将被送往被 L2 索引选中的组，并同时与组内的 8 个 Tag 地址进行比较，以确认是否命中。如果命中，则根据 Tag 比较的结果选择命中的行内容进行输出。否则，L2 Cache 将发起一次从片外存储器加载 Cache 行的过程。

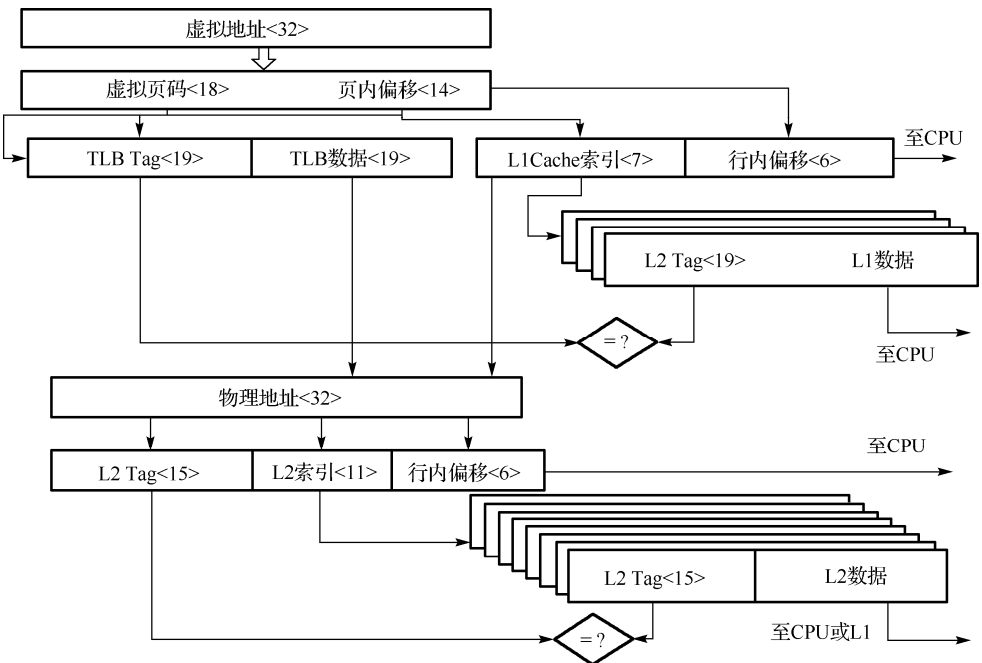


图 5-12 ARM Cortex A8 处理器的 Cache 架构

5.3 虚拟存储器

“Cache-主存”存储层次是为了提高访存速度，而“主存-辅存”存储层次是为了弥补主存容量的不足。早期的操作系统通过构建所谓虚拟存储器来实现主存与辅存之间的数据交换，其基本思想是将辅存映射为主存储器的一部分。在嵌入式系统中往往没有真正意义上传统 PC 以及大型机所采用的辅存机制（Secondary Storage，通常是磁盘系统），但由于操作系统的延续性（比如越来越多的嵌入式系统采用 Linux 操作系统，智能手机所采用的 Android 系统也是基于 Linux 系统的）以及对多任务运行环境的更高要求（比如，进程间的隔离），使得越来越多的嵌入式系统也对虚拟存储器（简称虚存系统）提出了要求。

5.3.1 虚拟内存技术的基本原理

早在 1961 年，英国曼彻斯特大学的 Kilburn 等人就已提出了虚拟存储器的概念。经过 20 世纪 60 年代到 70 年代初的发展完善，虚拟存储器已广泛应用于大型计算机系统。目前几乎所有的通用计算机都采用了虚拟存储系统，而在嵌入式领域，尤其是在移动智能终端应用中，越来越多的系统也开始采用虚存系统。

虚拟存储器是“主存-辅存”层次进一步发展的结果。它由价格较贵、速度较快、容量较小的主存储器和一个价格低廉、速度较慢、容量很大的辅助存储器（通常是硬盘）组成，在系统软件和辅助硬件的管理下，就像一个单一的、可直接访问的大容量主存储器。应用程序员可以用机器指令的地址

码对整个程序统一编址，就如同应用程序员具有对应于这个地址码宽度的存储空间（称为程序空间）一样，而不必考虑实际主存空间的大小。

计算机系统中经常同时运行多个进程，每个进程有自己的地址空间。如果给每个进程都分配一个与其全地址空间大小相同的主存区域，显然代价太高。更何况许多进程只用到其地址空间的一小部分。虚拟存储器把主存空间划分为较小的块（页面或段），并以块为单位分配给各进程。这样，多个进程就可以共享一个较小的主存空间。此外，大多数虚拟存储器还可以减少程序的启动时间，因为这时程序不需要像以往那样等到全部程序和数据调入主存后才能开始执行。

在非虚拟存储器中，当一个程序的大小超过主存空间的大小时，程序员就必须把程序划分为若干个能装得进主存的部分，并识别哪些部分（称为覆盖）在装入主存时可以相互覆盖。程序员还要控制这些覆盖在程序执行过程中的装入与重新装入（从辅存）。这就是所谓的程序覆盖技术。在这种方法中，程序员要保证程序不会访问主存空间大小以外的空间，而且要保证所需的覆盖会在适当的时机装入主存。显然，这种方法增加了编程的复杂度，降低了程序员的产出率。虚拟存储器解决了这个问题，它自动地对存储层次进行管理，免除了程序员的负担。这也是当时发明虚拟存储器的初衷。

虚拟存储器系统（通常虚存系统的软件部分属于操作系统内核的重要功能之一）将程序员从这种负担中解放出来，它自动管理一个由内存和辅存组成的两级存储系统。图 5-13 给出了虚拟存储系统的基本原理，假设一个占用了 4 个内存页（我们假设操作系统按照 4KB 的大小对存储器进行管理，一般将这个 4KB 的内存片段称为一个页）的程序，由于该程序使用虚拟地址，因此从这个程序的角度看它所拥有的 4 个页 A、B、C、D 是连续的。但在实际的内存页在物理地址空间的分布可能是不连续的，比如内存页 A 在虚地址中的地址是 0，而操作系统实际上却使用了一块物理地址为 16K 的内存页。而对于内存页 D，程序看到的是其被存放在内存页 C 之后（虚拟地址空间），而实际上该页的数据可能保存在磁盘中。

除了共享内存和自动管理存储器层次结构，虚拟存储器还使程序的加载变得更简单了。重定位技术可以使一个程序能够在物理内存的任何位置运行。所谓程序定位，是指将程序空间中给出的逻辑地址映像到主存的物理地址。程序重定位有静态重定位和动态重定位之分。静态重定位是指在程序执行之前，在装入或再装入的过程中，通过修改程序中的地址而完成地址空间变换。而在程序执行过程中，其在主存空间的位置就不能再改变。动态重定位是指只有在程序的执行过程中，真正访问指令和数据时才进行地址变换，产生物理地址。动态重定位使得同一程序可以很方便地装入主存中的任意一个位置执行。图 5-13 中的程序可以被放置在物理内存或是磁盘的任何地方，只需要改变映射关系就可以了。

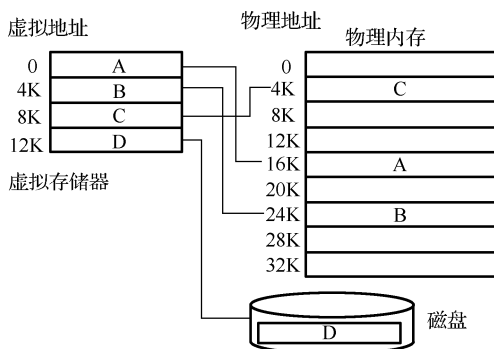


图 5-13 含有 4 个内存页的虚拟地址和物理地址的映射

虚拟存储器可以分为两类：页式（Page）和段式（Segment）。页式虚拟存储器把空间划分为大小相同的块，称为页面。常用页面的大小为 4KB~64KB。而段式虚拟存储器则把空间划分成可变长的块，

称为段。段的最小长度为 1 字节，最大长度因机器而异，通常为 $2^{16} \sim 2^{32}$ 字节。页面是对空间的机械划分，而段则往往按程序的逻辑意义进行划分。

采用页式虚拟存储器还是段式虚拟存储器对 CPU 有不同的影响。在页式虚拟存储器中，地址都是单一、固定长度的地址字，由页号和页内位移两部分组成。而在段式虚拟存储器中，地址需要用两个字表示，一个为段号，另一个为段内位移。这是因为段的长度是可变的。

采用虚拟内存技术就必须有对虚拟内存地址和物理地址进行转化的部件，同时还要提供各个进程的访问地址空间的保护等，这些主要功能就是由专门的内存管理单元（MMU，Memory Management Unit）来完成的。

MMU 模块的主要功能有：

- 进行地址译码，将程序中的虚拟地址转化成物理地址。
- 提供存储保护，使得不同进程之间的存储空间不会相互干扰和破坏。
- 实现和管理页表（Page Table）。页表中保存了虚拟地址与物理地址的映射关系，MMU 进行虚拟地址和物理地址转换时，最重要的步骤就是查找页表。

MMU 与 CPU 及主存间的连接关系如图 5-14 所示。

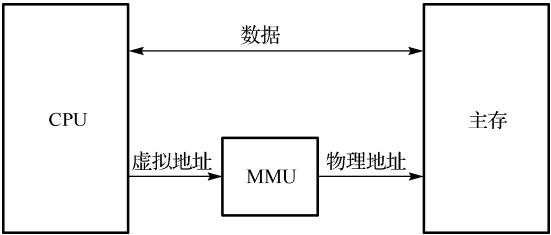


图 5-14 MMU 功能简图

5.3.2 虚实地址映射与转换

虚拟内存管理技术中所谓的地址映射就是把程序的虚拟地址空间映射到物理地址空间，即把用虚拟地址编写的程序放入物理内存中，并将虚拟地址与物理地址之间的对应关系建立起来。所谓地址变换指的是在程序实际运行时，将程序的虚拟地址转换成物理存储器能识别的物理地址。地址映射是依赖于地址转换技术实现的。地址转换技术根据页面的长度，将虚拟地址分为虚拟页号和页内偏移量两部分，如图 5-15 所示。同时，实际物理地址也被分为页帧号（Page Frame）和页内偏移量两部分。在地址映射过程中，建立了虚拟页号和页帧号的对应关系，而页内偏移量保持不变。操作系统将这种映射关系自动保存在页表中。

由于程序所使用的地址必须是连续的，所以编写程序时采用的是虚拟地址。在程序运行过程中，操作系统才将物理地址段分配给程序，这些物理地址段可以是非连续的。程序的虚实对应关系存放在页表中，也就是说每个程序都分别对应了一个页表，同时页表又存放在内存中，如果页表过大，还要对页表本身进行分页处理。每个页表项通常主要由索引、内容和一个有效控制位构成，索引就是虚拟地址的页号，而对应的物理页帧号就是内容，有效位表示当前页是否在内存中以及相应的控制信息。如果该页不在内存中，系统通过异常处理机制

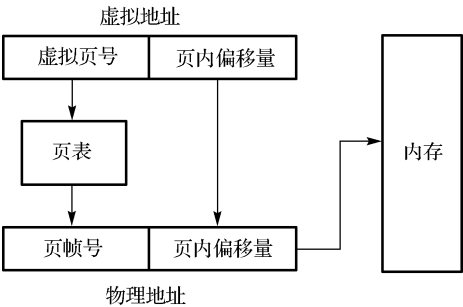


图 5-15 虚拟地址映射为物理地址

在存储器结构层次的下一级去寻找该页，然后调入内存，并分配相应的页表项，建立虚拟地址到物理地址的映射关系。每个进程都拥有自己独立的虚拟地址空间，因此每个进程都有自己独立的页表，一个进程的页表会伴随着该进程运行的结束（或上下文切换）而被下一个程序的页表更新（如图 5-16 所示）。

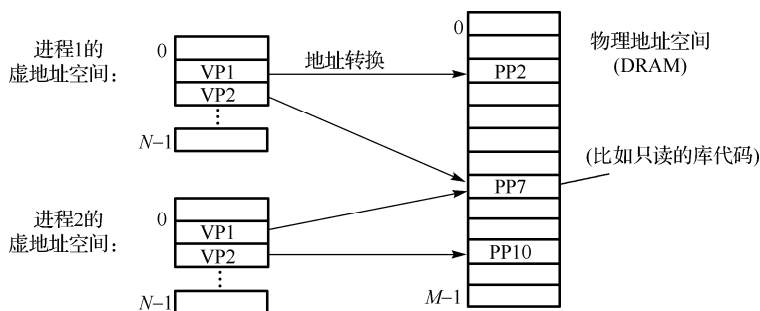


图 5-16 不同进程的虚地址空间

由于采用了不同的地址映射和转换方式，形成了 3 种主要的存储方式：基于页式的管理、基于段式的管理和基于段页式的管理。映射方式的选用会对 CPU 的处理造成影响。页式编址分成了页号和页内偏移两部分，但是只能编址定长的地址，地址位数不可变。段式编址只有单一的地址类型，只能给出段号，不能给出段的长度，而段页式编址是段式编址和页式编址的结合和改进。下面对这 3 种方式进行详细介绍。

1. 基于页式的管理方式

基于页式的管理方式将程序的虚拟地址空间分成若干个大小相等的块，每个块称为一个页面，并为各页加以编号，每个独立程序被分成一组相等的固定大小的页面的集合，每个页面都有自己顺序编号的虚拟页号。相应地，也将物理内存空间分成与页面大小相等的若干个存储块，称为物理页，也同样为它们加以编号。例如，某个程序的虚拟地址空间被分成 4 个页面，其各自的虚拟页号分别为 0、1、2、3，在为该程序分配内存时，以页为单位将程序中的若干个页分别装入多个可以不连续的物理页中，只要系统设置好相应的页面映射表（也就是页表），保存好虚实页面的对应关系，就可以根据给定的程序虚拟地址查询页面映射对应表，得到物理内存地址，实现对主存的访问。

页面其实是对虚拟地址进行逻辑上的划分，在基于页式的管理系统中的页面大小应适中。页面若太小，一方面虽然可以减小内存碎片，有利于提高内存的利用率，但另一方面每个程序占用的页面数就会增加，从而导致程序的页表过长，占用大量内存，还会降低页面置换的效率。如果页面较大，虽然能减小页表的长度，提高了页面置换速度，但又增大了页内碎片。因此，页面的大小选择非常重要，一般页面大小应为 2 的幂，为 512B~8KB。

基于页式的管理方式中程序的起点一定是页面的起点，虚拟页号和页内偏移量组成了用户程序的虚拟地址。由于虚拟地址和物理地址的页面大小通常是一样的，所以页内偏移量在记录虚拟地址和物理地址的映射时是不需要保存的，只需保存虚拟页号和页帧号即可。同理，只用页帧号就可以代替页表中装入主存起点的地址。这样映射表硬件就可以大大简化，并能加快查表速度。图 5-17 表示出了基于页式管理系统的虚实地址变换过程。存储器管理单元 MMU 将页表起始地址与虚拟页号 3 相加得到将要访问的页表地址 3，访问页表得到该页的物理块号 5，物理块号加上虚拟地址的页内偏移量字段就是物理地址，这样就完成了从虚拟地址到物理地址的变换。注意：①通常情况下页表项中并没有存储虚拟页号，因为页表是按照虚拟页号的顺序组织的，可以通过虚拟页号+页表首地址直接找到对应的物

理页号（类似于数组直接通过下标进行寻址）；②每个进程都有自己的页表，因此当发生进程切换时，操作系统需要负责更新页表首地址寄存器的值。

每个页表项中除了包括对应的物理页框地址外，还需要一些控制信息：①有效位，用于表示该页是否在物理内存中；②替换信息，用于支持相应的替换策略；③“脏位”，用于表示该页是否被修改过，如果被修改过，则该页被换出时需要回写到辅存；④保护信息位，用于实现对该页的访问控制与保护。

基于页式的管理方式主要有以下优点：

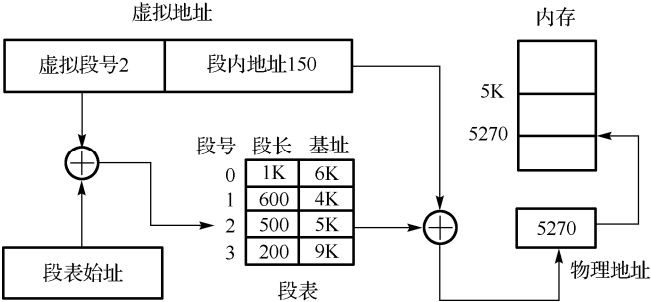
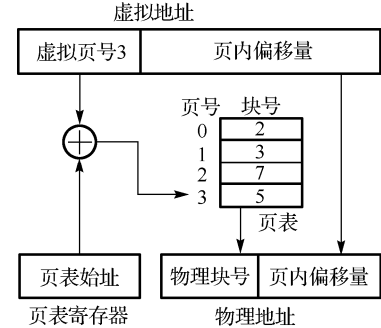
- （1）基于页式的管理方式提高了主存的利用率。每个程序只有大约半页的浪费，而基于段式的管理方式中任意两个程序之间都要浪费一定空间
- （2）基于页式的管理方式简化了页表。与基于段式的管理方式相比，基于页式的管理方式只需保存较少的字段，某些关键字的长度也较短。
- （3）基于页式的管理方式提高了地址映射和转换的速度。

然而该方式也存在两个主要缺点：

- （1）不利于程序的模块化。程序被划分为固定大小的页，而通常情况下很难保证一个页面包含整个程序段，所以某些程序段就会被拆分，然后放入相邻的几个页。
- （2）页面较小且大小固定，所以页表会比较庞大，占用的存储空间也大。

2. 基于段式的管理方式

通常，程序都是根据其内容的逻辑关系分成若干个程序段，每段的大小各不相同。基于段式的存储管理系统就是按照这个原理将物理内存空间分成大小不同的段，每个段都从 0 开始编号，并为每个段分配一个连续的地址空间，而且各个程序段可以离散地分布在内存的不同分区中。同样，虚拟地址由虚拟段号和段内地址组成。为使程序正常运行，只要能从物理内存中找到每个逻辑段所对应的位置，就在系统中给每个程序建立一张段映射表，简称“段表”。图 5-18 是基于段式管理系统的虚实地址变换过程。基址寄存器保存有段表的起始地址，将段表起始地址加上虚拟段号 2 就得到程序段的段表地址 2，读段表得到基址 5K，将基址与段内地址 150 相加得到物理地址 5270（ $5 \times 1024 + 150 = 5270$ ），这样就完成了段式管理方式下的虚实地址变换。



基于段式的管理方式具有以下优点：

- （1）基于段式的管理方式有利于程序的模块化。由于分段是根据程序的逻辑结构进行划分的，故

某一个程序段的改变不会影响其他段。

(2) 基于段式的管理方式有利于程序实现信息共享。因为程序和数据共享是以信息的逻辑单位为基础的，而段正是信息的逻辑单位。

(3) 基于段式的管理方式有利于实现信息的包含。信息保护同样是对信息的逻辑单位进行保护，因此，段式的管理方式能更有效和方便地实现信息保护功能。

(4) 基于段式的管理方式有利于程序的动态增长和动态链接。在实际应用中，比如数据段在使用过程中会不断增长，而事先又无法确切地知道数据段会增长到多大，段式的管理方式却能较好地解决这个问题。由于程序段数据块的独立性，程序运行前，可以根据需要以程序段为管理单位装入内存。

同样，基于段式的管理方式也具有以下缺点：

(1) 内存的利用率较低。由于每个程序段的长度不同，故段的起止点不能确定，因此内存分配比较复杂，而且在每个段间还容易产生碎片，造成存储空间浪费。

(2) 虚实地址变换更加耗时，这是因为段表比页表复杂。

3. 基于段页式的管理方式

基于段页式的管理方式就是将基于页式和段式的管理方式结合起来，它集成了两者的优势。其基本原理是：在虚拟地址上，对程序采用基于段式的管理方式进行逻辑单位上的分段，再对每个程序段采用基于页式的管理方式进行分页；在物理内存上，将物理内存空间分成与页面大小相等的若干个物理块。程序对于内存上的调度都是以页为基本单位的，这样既能获得内存的高效利用，又能按照程序段实现信息保护和共享。同样，这种管理方式也存在缺点，在进行虚实地址变换时，需要进行多次查表，需要耗费更多的时间。

在基于段页式的管理系统中，每个程序对应一个段表，每个程序段对应一个页表。段表的每个表项分别对应一个程序段，表项中指明了该段的页表的起始地址和该段相应的控制保护信息。页表指明了程序段中页面在内存中的位置。

如图 5-19 所示，要得到物理地址，需要进行两次查表：第一次查询段表（将段表寄存器中的段表起始地址加上虚拟地址的虚拟段号，得到页表起始地址），第二次是查询页表（将页表起始地址与虚拟地址的虚拟页号相加，得到物理块号），最后将物理块号加上虚拟地址的页内偏移量字段，得到物理地址，这样就完成了基于段页式管理方式下的虚实地址变换。

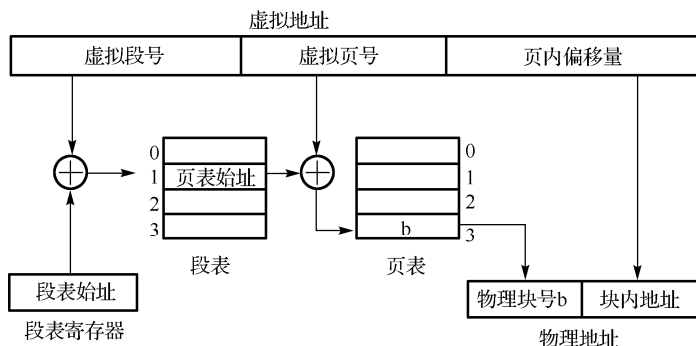


图 5-19 基于段页式管理的虚实地址变换

5.3.3 快速地址转换技术

页表通常存放在主存储器中，因此程序要从主存储器中访问一个数据（取指令、读数据或写结果）

至少要访问主存两次：第一次访存获得物理地址（这个过程实际就是查找页表的过程），第二次访存获得数据。由于主存储器的速度比 CPU 慢得多，每次都这样做效率是非常低的。

由于程序在执行过程中具有局部性的特点，因此对页表的访问并不完全是随机的。在某段时间内，对某些特定地址访问比较频繁。根据这一特点，可以把经常访问的页表项缓存在一个小容量的高速存储器内，称为“快表”（即地址转换旁路缓冲存储器，TLB，Translation Lookaside Buffer）。当快表中查不到时，再从存放在主存储器中的页表中查找实页号。与快表相对应，存放在主存储器中的页称为慢表。慢表是一个全表，快表只是慢表的一个副本，而且只存放了慢表中很少的一部分，可以认为快表实际上就是页表的高速缓存（Cache）。TLB 就存放在存储管理单元 MMU 中。TLB 的使用大大加速了虚实地址之间的转换。TLB 作为页表的 Cache，加速了虚地址和物理地址的转换过程，而传统的指令 Cache 和数据 Cache 则加速了处理器访问存储器的过程。

对每一次访问，我们都将待转换的虚地址的高位（也就是虚页号）连接当前正在运行的进程的进程号以形成一个独一无二的页地址，将其与 TLB 中的各表项进行比较查找。如果命中，则物理号就用来形成地址，并且设置相应的访问位。如果处理器在执行写操作，那么还要设置修改位。如果查找 TLB 发生缺失，就必须访问存放在主存中的慢表进行查找，同时 MMU 还需要判断缺失的页在不在主存中。如果该实页在主存中，那么把查到的实页号送入主存储器的地址寄存器，并把这组虚实页号映射关系缓存到快表中。若快表已满，则要采用某种替换算法，替换掉其中一项。如果该实页不在主存中，那么本次 TLB 缺失就是一次真正的缺页（此时操作系统将通过访问辅助存储器，通常也就是硬盘，将保存在磁盘中的页读入可分配的主存储器中，并修改页表中虚地址和实地址的映射关系）。一般来说，TLB 中缓存的页表项比主存中的页表项数少得多，发生 TLB 缺失会比发生真正的缺页频繁得多。

在页表中除了存储物理页号外，通常还存储一些额外的控制位信息，用于控制哪些访问被允许。最简单的例子就是写操作允许位，该位决定此物理页面是否允许写操作。

与指令 Cache 和数据 Cache 一样，在 TLB 内进行地址比较时是将所有的表项和输入地址（虚页号）同时做比较（对于采用全关联的 TLB）。这种比较方法效率非常高，但是硬件代价很大。与 Cache 架构类似，在很多处理器设计中，通常 TLB 也采用两级缓存结构：第一级 TLB 比较小，但是采用全关联方式进行映射；第二级 TLB 容量较大，但采用组关联方式进行映射。另外，对于采用分离式 Cache 结构的 CPU 而言，其 TLB 一般也分为指令 TLB 和数据 TLB，用于分别缓存取指与数据访问的页表项。

5.3.4 地址保护机制

正如前面所介绍的，虚拟存储器主要的作用包括两个：①使得程序员可以使用比物理存储器更大的内存空间；②实现访问控制和保护。保护机制必须保证尽管有多个进程共享同一段主存，但一个进程不能往另一个用户进程或者操作系统的地址空间里写入数据。页表中的写访问位可以防止一页被改写。在某些情况下也要防止一个进程读取另一个进程的数据。如图 5-20 所示，进程 i 和进程 j 都映射了物理空间 PP6，但是进程 i 对于该物理页只可读不可写，而进程 j 则可读可写。

每个进程有它自己的虚拟地址空间。因此，如果操作系统管理页表的组织，使独立的虚拟页被映射到不同的物理页上，一个进程就无法访问另一个进程的数据。当然，这也要求用户进程无法改变页表的映射。如果操作系统能防止用户进程更改自己的页表，就能保证安全性，然而这样一来操作系统就必须负责更改页表。把页表放在操作系统的地址空间就能满足所有要求。

为支持操作系统实现地址保护，CPU 硬件至少需要提供以下 3 种能力：

（1）支持至少两种模式，分别指出正在运行的进程是用户进程还是操作系统进程。操作系统进程也称为核心（Kernel）进程、管理（supervisor）进程或执行（executive）进程。通常情况下，现代处理器支持不同的运行模式，比如 ARM 处理器支持所谓的特权模式与非特权模式，用户进程运行在非

特权模式（也就是 ARM 处理器的 User 模式）。而操作系统运行在特权模式（对于 ARM 处理器而言，几乎所有的特权模式都与中断有关，比如 IRQ 模式、FIQ 模式、SVC 模式、Abort 模式、Undef 模式等）。运行于用户模式的进程是无法通过指令（软陷或软件中断指令除外）的执行切换到特权模式的，唯一可以从用户模式进入特权模式的方法是中断的发生，而操作系统将接管所有的中断处理（这也是为什么塔利鲍姆在他的《现代操作系统》一书^[4]中指出“中断是操作系统的入口”的原因）。当 CPU 处于特权模式时，操作系统软件可以读取和修改程序状态字（PSR）的所有内容，并可以运行只能在特权模式才能被执行的指令，比如协处理器指令（就 ARM 处理器而言，对于 Cache 和 MMU 的配置与管理都是通过协处理器指令来完成的。）

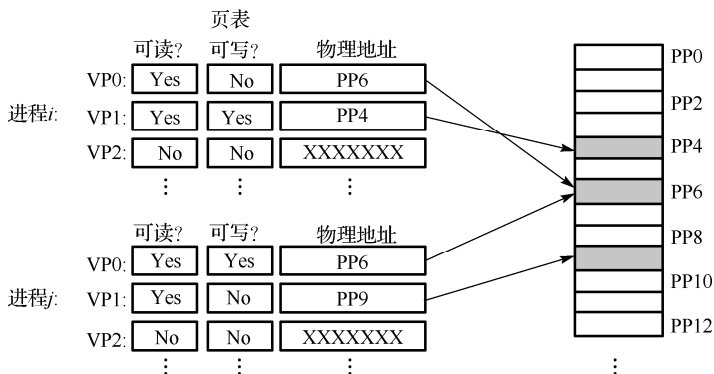


图 5-20 地址保护机制

(2) 提供一部分用户进程只能读不能写的 CPU 状态。这包括指出处理器是处于用户态还是管理态的用户/管理模式位，以及页表指针和 TLB。操作系统利用只能在管理模式（或特权模式）使用的特殊指令来对它们进行写操作。

(3) 提供能让 CPU 从用户模式到管理模式相互转换的机制。从用户模式到管理模式的转换一般是由系统调用异常处理（也就是软件中断指令，对于 x86 处理器而言是 INT 指令，对于 ARM 处理器是 SWI 指令，对于 68000 处理器而言是 TRAP 指令）完成的，它以特殊指令的方式出现，把控制权转到管理代码空间的指定位置。和其他异常处理一样，系统调用处的程序计数器的值被保存起来，CPU 也被置于管理模式。从异常中返回到用户模式使用的是从异常中返回的指令，将恢复产生异常时进程的状态。

通过利用这些机制并把页表保存在操作系统的地址空间，操作系统就可以建立并管理页表，并防止用户进程改变它们，确保用户进程只能访问由操作系统提供给它的主存地址段。也就是说，只有操作系统才可以改变地址变换。

当进程希望以受限的方式共享信息时，操作系统必须支持它们，因为访问另一个进程的信息要求改变被访问进程的页表。写访问位可以用来限制共享为只读的，并且与页表的其他部分一样，这个位只允许被操作系统更改。为了允许另一进程 P1 读取进程 P2 所在的一页，P2 将通过系统调用请求操作系统在 P1 的虚拟地址空间里为一个虚拟页生成页表项，指向 P2 所想要共享的物理页。如果 P2 要求，操作系统可以使用写保护位防止 P1 对数据进行改写。由于仅在发生 TLB 缺失时访问页表，所以任何决定了页的访问权限的位不仅要包括在页表中还要包括在 TLB 中。

当操作系统决定从执行进程 P1 变换到执行进程 P2（称为上下文切换 context switch，或进程切换 process switch）时，它必须确保 P2 无法访问 P1 的页表，因为那样将危及到保护。如果没有 TLB，它只需要把页表基址寄存器转而指向 P2 的页表（而不是 P1 的）就足够了。有了 TLB，则必须清除 TLB

中属于 P1 的表项以装入 P2 的表项。若操作系统又切换回 P1，则又需要通过 TLB 缺失来重新装入 P1 对应的表项。如果进程切换非常频繁，效率就会很低。产生这个问题是因为 P1 和 P2 使用的是同一个虚拟地址空间，为了防止混淆必须更新所有 TLB 表项。

为了减少进程切换时需刷新所有 TLB 表项的开销，通常的做法是增加一个进程标志符（地址空间 ID，简称为 ASID）来扩展虚拟地址空间。每个进程里的任何地址都被进程的 ASID 扩展，从而产生唯一的地址。这个字段标识了正在运行的进程，当操作系统切换进程时它就保存在由操作系统装入的寄存器中。进程标识符与 TLB 的标记部分相连接，因此 TLB 命中只有在页号与进程标识符相匹配时才会发生。这样就不必更新所有 TLB 表项了。

5.3.5 处理缺页和 TLB 缺失

当 TLB 命中时，TLB 把虚拟地址转换成物理地址是很简单的。处理 TLB 缺失和缺页要复杂得多。当 TLB 中没有一项与虚拟地址相匹配时就发生 TLB 缺失。TLB 缺失有两种可能：

- 实页在主存储器中，这是一次普通的 TLB 缺失。
- 实页不在存储器中，称为缺页。这种情况将花费很大的时间代价。

对于第一种情况，CPU 可以采用硬件机制或者软件机制进行处理。对于采用硬件机制的 CPU（比如 x86 处理器和 ARM 处理器），MMU 将负责查找页表，并通过硬件将相应的页表项填充到 TLB 中，如果 TLB 已满，则需要按照一定的替换算法替换某个 TLB 表项。这种情况一般时间代价不大。不管是通过硬件处理还是通过软件处理，只需要短短的几步操作，就可以把一个匹配的页表项从主存复制到 TLB 中。如果采用软件机制处理 TLB 缺失（比如 MIPS 处理器、Alpha 处理器和 UltraSPARC 处理器），则硬件会发起一个异常（Exception），操作系统接管这个异常，并负责查找页表，将找到的页表项插入 TLB 中。采用硬件机制的优点包括：不需要产生异常；其他指令可以继续执行；没有其他的指令和数据会被装载到 Cache 中。其缺点主要是灵活性较差，因为页表的结构需要固化在硬件逻辑中（否则，硬件将无法直接访问页表），由此操作系统的设计必须迁就硬件。采用软件机制的优点包括：操作系统可以定义页表的结构；可以采用更加高级的 TLB 替换策略。但其缺点是必须产生异常以使操作系统接管该事件，由此其性能开销是比较大的。另外，由于异常处理程序的执行，使得新的指令和数据会被装载到 Cache 中。

第二种情况即处理缺页，通常需要使用异常机制中断正在运行的进程，把控制权交给操作系统，待操作系统将辅存中的页调入主存并更新页表后恢复执行被中断的进程。

一旦操作系统知道了引起缺页的虚拟地址，它必须完成以下 3 个步骤：

- 用虚拟地址在慢表中查找，并在磁盘上找到被访问的页的位置。
- 选择被替换的物理页，如果所选页被重写过，则必须在把新的虚拟页装入之前将该物理页写回磁盘上。
- 启动读操作，把被访问的页从磁盘上调到所选择的物理页的位置上。

最后一步将花费数百万个时钟周期（如果被替换页被重写过，第二步也将如此），因此，通常操作系统会选择另一个进程运行，而挂起当前产生缺页的进程，直到磁盘访问结束。

对于嵌入式设备而言，一般很少采用辅存（不管是磁盘还是 Flash 存储器）存放内存页，因此也很少会产生类似于 PC 因为虚拟内存页换入换出而导致的频繁磁盘操作。

5.3.6 ARM Cortex A 系列处理器的虚存管理

在 ARM 系统中，MMU 主要完成以下几个工作：

- 虚拟地址到物理地址的映射，在访问内存之前，通过 MMU 中的地址重定位功能转换得到物理地址。
- 存储器访问权限的控制，用户程序只能访问自己内存区域的数据，访问的每页都需要设置特定的读写权限。
- 设置虚拟存储空间的缓冲特性，即旁路转换缓冲器 TLB，存放最近被访问的页的转换信息。

MMU 硬件中用多个重定位寄存器来支持虚实地址的转换，使用页表存放描述系统中用到的虚拟存储器映射的数据。图 5-21 是页、MMU 以及页帧之间的关系，页是实现上述这些功能的重要手段，每个页表项对应于虚拟存储空间的一个页，该行包含了该虚拟内存页对应的物理内存页的地址、分配给该页的访问权限和该页的缓冲特性等。

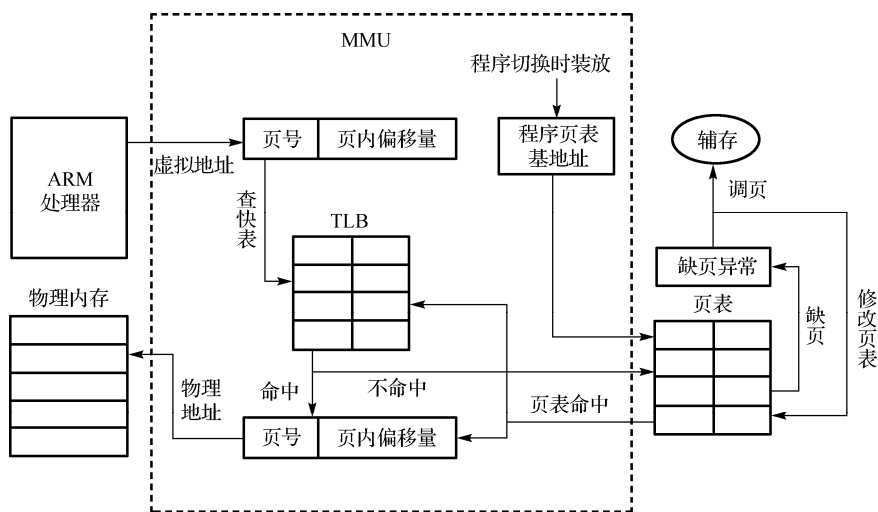


图 5-21 ARM MMU 存储系统

ARM 访问内存时，MMU 首先要查找 TLB 中的虚拟地址表，如果 TLB 中没有相应的映射关系，则转换表遍历硬件，从内存中的页表中获取转换和访问权限，一旦命中，该页表项就被放到 TLB 中，它会放在一个没有使用的入口处或者替换一个已有的页表项（取决于 TLB 的替换策略）。而一旦得到相应的页表项后，这些信息将被用于处理以下工作：①C（高速缓存）和 B（缓冲）位将被用来分别控制高速缓存和写缓冲，并决定是否进行高速缓存；②访问权限和域位用来控制是否允许访问。如果不允许，则 MMU 会向 ARM 处理器发出一个访问异常，否则访问将被允许。

Cortex A9 处理器的虚存系统符合 ARM 公司 VMSAv7（Virtual Memory System Architecture version 7）体系结构的定义，其页大小可以支持 4KB、64KB、1MB 和 16MB。其 MMU 页表项支持全局与地址空间 ID，使得进程上下文切换时不需要刷新整个 TLB。Cortex A9 处理器支持两级 TLB，其中第一级微 TLB（uTLB）分为 32 入口或 64 入口的指令 uTLB 以及 32 入口的数据 uTLB，uTLB 采用全关联方式进行映射。第二级主 TLB 为指令数据统一的 2 路组关联结构，可配置为 64、128、256 甚至 512 个入口。对于 uTLB 的访问可以在一个时钟周期内完成，而对于主 TLB 的访问可能需要多个时钟周期。当主 TLB 发生缺失时，硬件将发起对页表的查询，并将最新的页表项更新到 TLB 中。由于采用 ASID 区分不同进程的页表项，在发生进程上下文切换时，只需要刷新 uTLB 的内容，而主 TLB 的内容可以不用刷新。

Cortex A9 处理器采用两级页表进行地址转换，之所以采用两级页表的设计是为了能够减少只用一级页表所需要占用的存储空间（对于采用 4KB 页的系统，一个进程的 4GB 虚拟地址空间需要 1M

个页表项，如果每个页表项占用 32 位，那么一个页表的大小就会达到 4MB）。Cortex A9 的一级页表将 4GB 内存空间分为 4096 个 1MB 的段，这样一级页表将有 4096 个入口，每个页表项占用 32 位。每个页表项中的内容有 4 种可能（通过页表项最低两位进行区分），如图 5-22 所示：①该虚拟地址对应的 1M 段未被映射到物理存储器，也就是缺页（段）；②该虚拟地址需要经过二级页表转换，因此页表项中保存的是这个地址段对应的二级页表基址；③该虚地址对应的 1M 段已被映射到物理存储器，CPU 可以获得页表项中的 1MB 段物理基址；④该页表项中保存了 16M 超段的物理基址^①。对于第 3 种和第 4 种情况，页表项中还保存了该内存段的访问保护位，比如是否允许访问（AP 位）、是否允许 Cache 缓存（C 位）等。这种两级页表的管理机制类似于前面 5.3.2 节介绍的段页式管理。

缺页	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	忽略																												0	0		
二级页表	二级页表基址																						P	Domain			DNZ		0	1		
1MB段	段基址												S B Z	0	n G	S	A P X	TEX	AP	P	Domain			X N	C	B	1	0				
超级段	超段基址								SBZ					1	n G	S	A P X	TEX	AP	P	Domain			X N	C	B	1	0				
保留																													1	1		

图 5-22 Cortex A9 处理器的一级页表结构

对于一级页表的访问过程如图 5-23 所示，CPU 发出的虚拟地址的高 12 位作为偏移量加上一级页表基址寄存器中存放地址的高 18 位构成该虚拟地址对应的一级页表项地址。如果该表项中存放的是二级页表基址，则访问二级页表；如果该表项中存放的是 1MB 段所对应的物理基址（12 位），则该基址和虚拟地址的低 20 位就可以构成 1MB 段的物理地址。

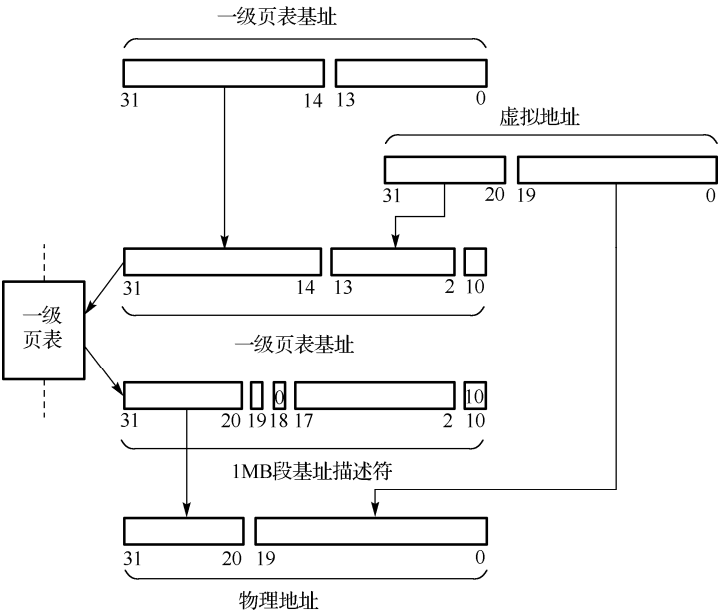


图 5-23 通过一级页表获得 1MB 段物理基址的过程

① 注意：区分 1MB 段还是 16MB 超段，需要用到页表项的第 18 位，该位为“0”表示 1MB 段，该位为“1”则表示超段。

每个二级页表包含 256 个页表项，每个页表项占用 4 字节，因此一个二级页表占用 1KB 的空间，并且二级页表的存放地址必须按照 1KB 对齐。通过页表项的最低两位可以区分页表项的内容，如图 5-24 所示：① 缺页，对该地址段的访问将产生 Data Abort 或者 Prefetch Abort；② 4KB 小页的物理基址；③ 64KB 大页的物理基址。当然，页表中还包括了访问控制位，有兴趣的读者可以参阅扩展阅读[2]。

缺页	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	忽略																														0	0
大页	大页物理基址																X N	TEX		n G	S	A P X	SBZ		AP	C	B	0	1			
小页	小页物理基址																n G	S	A P X	TEX		AP	C	B	1	X N						

图 5-24 Cortex A9 的二级页表结构

图 5-25 给出了 Cortex A 系列处理器采用二级页表进行地址转换的全过程。

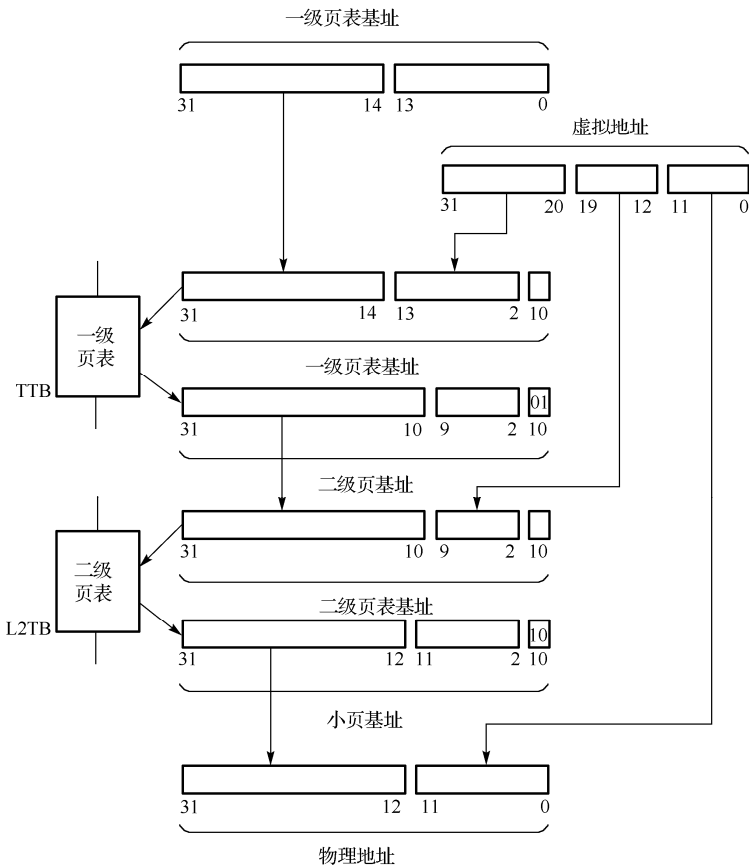


图 5-25 Cortex A 系列处理器的二级地址转换过程

另外，为了防止进程切换时需要整个刷新 TLB，Cortex A 处理器在每个 TLB 表项中增加了 ASID。所谓 ASID 其实是一个 8 位数，它可以是用户进程号的一部分（通常进程 ID 是一个 32 位数），MMU 中设计了一个专门的硬件寄存器，用于存放当前进程的 ASID，CPU 发出的虚拟地址(图 5-26 中的 VA)，以及当前的 ASID 会一起被送往 TLB 进行查找，只有同时符合的表项中存放的描述符才是有效的，并最终得到该虚地址所对应的物理地址。

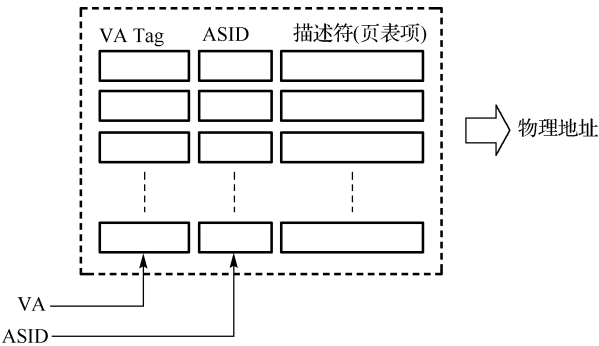


图 5-26 Cortex A 系列处理器的 TLB 结构

图 5-27 说明了如何通过 ASID 来避免 TLB 刷新，在 TLB 中保留了一些全局的 TLB 表项（一个表项是否为全局是通过页表项中的控制信息位来决定的，见二级页表中的 nG 位，如果该位置位则表示该页内容不是全局页，换言之，该页属于某个进程），这些表项所表示的内存页为进程间共享的，因此不要在任务切换时进行刷新。而对于非全局页，如图中所示，任务 A、B、C 都拥有各自虚拟空间的 0x000 地址，如果没有 ASID 进行区分，唯一的解决办法就是任务发生切换时刷新整个 TLB，显然这是一个低效的解决方案。Cortex A 系列处理器在 CP15 c13 寄存器中保存了当前任务的 ASID，如图中所示，当前该寄存器中保存的 ASID 标示当前任务为 C，因此结合该 ASID 和 CPU 发来的虚地址 0x000，我们可以在 TLB 中找到任务 C 的零地址所对应的物理地址。

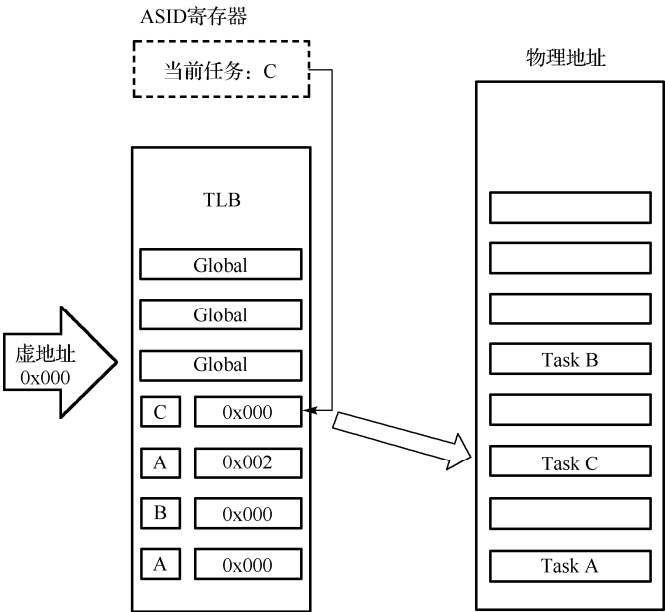


图 5-27 通过 ASID 避免 TLB 刷新

5.4 片外存储器

一个存储器包含许多存储单元，每个存储单元可存放一个字节。每个字节存储单元的位置都有一个编号，即地址，一般用十六进制表示。一个存储器中所有存储单元可存放数据的总和称为它的存储

容量。假设一个存储器的地址码由 20 位二进制数（即 5 位十六进制数）组成，则可表示为 2^{20} ，即 1M 个存储单元地址。每个存储单元存放一个字节，则该存储器的存储容量为 1MB，如图 5-28 所示。

对于嵌入式系统开发的程序员来说，片上存储器中只有 SPM 可以通过存储单元地址来访问，Cache 对程序员是透明的，CPU 中的寄存器只有在写汇编语言时才会去直接读写寄存器。对于片外存储器，一般是通过存储器单元地址来访问存储单元的内容。

地址	存储单元	
00000 _H	0011	0100
00001 _H	0101	1101
00002 _H	0111	0110
00003 _H	0011	1011
⋮	⋮	⋮
FFFFD _H		
FFFFE _H		
FFFFF _H		

图 5-28 存储单元地址与存储内容示意图

CPU 根据地址访问存储单元，读出或写入数据。从一个存储单元读出或写入数据的时间称为读写时间，两次读/写操作之间的间隔称为存取周期。这两项是衡量存储器存取速度的指标。

- 读出时间：从存储器中读出数据所需要的时间，等于从发出读请求到数据被传送到输出端上的有效时间，如图 5-29 所示。
- 写入时间：从提出写请求到最终把输入数据写入存储器之间所经过的时间，如图 5-29 所示。
- 读/写周期时间：在前后两次读或写操作之间所要求的最小时间间隔。这一时间通常大于存取时间，如图 5-29 所示。

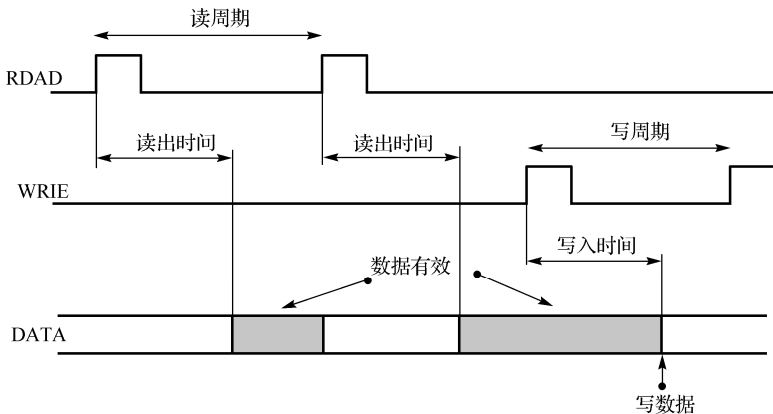


图 5-29 存储器的读写时序

对于一些嵌入式系统中常用的存储器，需要了解存储器的存储特点、引脚和访问时序。SRAM 和 Nor Flash 的引脚和读访问时序是类似的（Nor Flash 的写时序需要特殊协议）。SDRAM 和 DDR2/3 的引脚类似，访问时序稍有差别。NAND Flash 和 SD/MMC 存储卡都是基于闪存的存储设备，需要按照规定的方式和协议来访问。

5.4.1 静态随机存储器（SRAM）

易失性指的是断电后保存在存储器上的数据会消失，即电源关闭时无法保留数据。如果需要保存数据，就必须把它们写入可以长期保留数据的存储设备，即非易失性存储器中。易失性存储器的主要类型为随机存取存储器，又称作随机存储器（RAM）。所谓随机存取，指的是读取或写入存储器信息所需要的时间与这段信息所在的位置无关。相对地，读取或写入顺序访问（sequential access）存储器时，所需要的时间与位置则有关系（比如早期计算机使用的磁带存储器）。

随机存取存储器可以说是所有存储器中写入和读取速度最快的。RAM 可以进一步分为 SRAM（静

态随机存储器）和 DRAM（动态随机存储器）两大类。这两类存储器保存数据的机制是不同的，前者是利用双稳态电路保存信息，而后者则是利用电容存储信息。通常情况下，SRAM 具有快速访问的优点，但生产成本较为昂贵。

SRAM 的组成结构在讲解 Cache 的基本组成时（5.2.1 节）介绍过，本节介绍 SRAM 的读写时序，便于后面介绍外部存储器控制器 EMI 的设计。SRAM 一般用作片上存储器（Cache 或者 SPM，采用 SRAM 作为片上存储器的一个重要原因是因为 SRAM 的制造工艺与 CMOS 工艺是兼容的，也就是说可以方便地采用相同的工艺同时制造 CMOS 逻辑电路和存储器电路），也可以用作片外存储器。SRAM 读时序如图 5-30 所示，写时序如图 5-31 所示。图中各个引脚信号的意义如下。

- CS：存储器的片选信号，通常和 SoC 的 EMI 模块的存储器片选信号线相连接，CS 有效时（图中为低电平有效），表示 SoC（或处理器）选中该存储器，即 SoC 的地址总线上给出的地址在该存储器的存储空间范围内，SoC 可以访问该存储器。
- Address：SRAM 存储器的地址线，用于选择访问存储器中的哪个存储单元，通常对应连接到 SoC 芯片的地址总线上。
- DQM：地址锁存信号，DQM 有效时（图中为低电平有效），表示此时 SoC 上给出的是有效的地址信号。
- OE：存储器的读信号，通常和 SoC 芯片的 EMI 模块提供的存储器读信号线相连接，OE 信号有效时（图中为低电平有效），表示 SoC 芯片的 EMI 模块准备读取该存储器的某个存储单元的数据。
- WE：存储器的写信号，通常和 SoC 芯片的 EMI 模块提供的存储器写信号线相连接，WE 信号有效时（图中为低电平有效），表示 SoC 芯片的 EMI 模块准备往该存储器的某个存储单元写数据。
- Data Out：读数据总线，通常和 SoC 芯片的读/写数据总线相连接，SoC 芯片的 EMI 模块可以从存储器的读数据总线上读出欲访问存储单元的数据。
- Data In：写数据总线，通常和 SoC 芯片的读/写数据总线相连接，SoC 芯片的 EMI 模块可以从存储器的写数据总线上往欲访问的存储单元中写入数据。

图 5-30 和图 5-31 中各个时序参数的意义如下，这些时序参数值对于不同厂商提供的存储器来说是不同的，通常 SoC 的外部存储控制器（EMI）中会提供相应的配置寄存器，使得用户可以根据系统所选择的存储器芯片参数配置 EMI 的时序（程序员需要仔细阅读所使用的存储器时序文档，并根据文档所给出的参数配置 EMI，否则可能造成读写错误）。

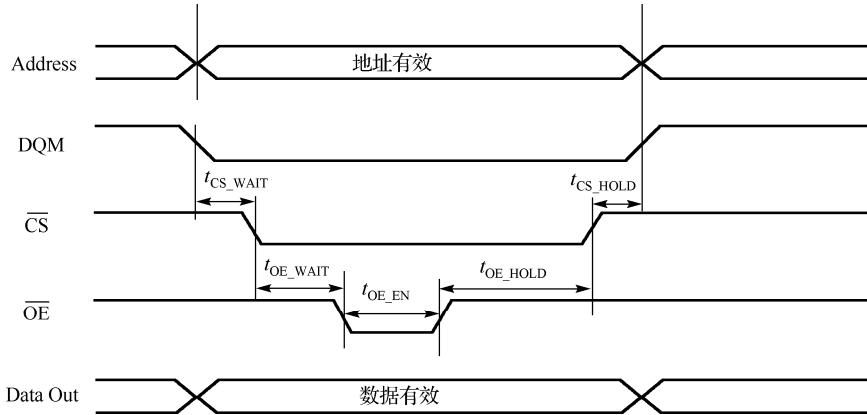


图 5-30 SRAM 的读时序

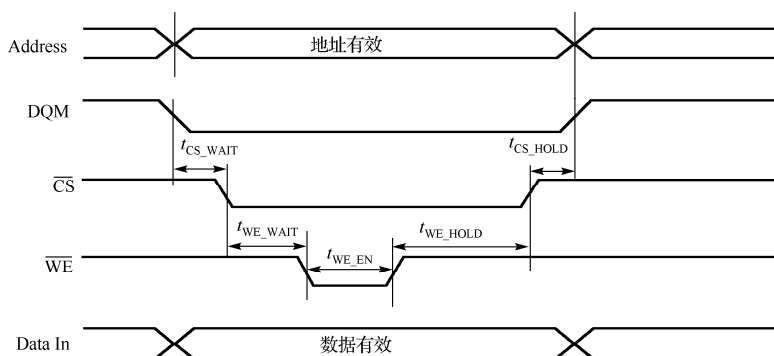


图 5-31 SRAM 的写时序

- t_{CS_WAIT} : 地址有效后片选保持为高电平（无效）的最短时间。
- t_{CS_HOLD} : 片选从低电平变为高电平之后保持高电平的时间。
- t_{OE_WAIT} : 片选有效后 OE 保持高电平所需要的时间。
- t_{OE_HOLD} : OE 信号从低电平到高电平后保持高电平所需要的时间。
- t_{OE_EN} : OE 信号保持低电平（有效）需要的时间。
- t_{WE_WAIT} : 片选有效后 WE 保持高电平需要的时间。
- t_{WE_HOLD} : WE 信号从低电平到高电平后保持高电平所需要的时间。
- t_{WE_EN} : WE 信号保持低电平（有效）需要的时间。

通过以上的时序分析，我们可以知道 SRAM 的读写时序本质上是异步的问答模式。也就是说，SRAM 的读写并不需要 CPU（或者是 SoC 的 EMI，也就是 SoC 的外部存储控制器）和存储器之间的时钟同步，EMI 只是发出地址，给出相应的控制电平信号（CS、OE/WE 等），等待存储器芯片规定的时间，然后从数据总线上锁存存储器输出的数据（读操作）或者是保证 EMI 发出的数据在总线上停留相应的时间（写操作）即可。待本次读写完成后，才可以开始下一次的读写周期。CPU 或 EMI 发出一个地址，得到一个数据（通常只有 8 位、16 位或者 32 位），然后再发出下一个地址，再得到一个数据。SRAM 的读写时序对于现代处理器的 Cache 行填充（通常至少一次发起 4×32 位传输），或者是 DMA 批量数据传输而言是非常保守而低效的。这也是为什么现在嵌入式系统的高性能处理器都选用 DRAM 作为主存储器的原因之一（另一个原因是 SRAM 的单字节成本要比 DRAM 高得多）。

5.4.2 动态随机存储器（DRAM）

动态随机存储器（DRAM, Dynamic Random Access Memory）由于性价比很高，且扩展性也不错，所以被现代计算机系统（从大型机到 PC 再到手机类的嵌入式系统）大量地采用作为系统主存储器。DRAM 的数据按比特位存储在极小的电容上（通常该电容是 MOS 晶体管的栅漏电容），因此 DRAM 的容量密度要比逻辑门组成的 SRAM 大得多。1MB 的片上 SRAM 对于嵌入式 SoC 来说算是比较大的，而使用 DRAM 的嵌入式 SoC 通常带有 256MB 以上的 DRAM 存储器，甚至可以高达 2GB 以上。

DRAM 的存储密度比 SRAM 要大得多（SRAM 需要 6 个晶体管存储 1 比特，DRAM 只需要 1 个晶体管），因此每比特的价格要低得多。然而，DRAM 的读取速度通常比逻辑门组成的存储器要慢。另外，由于 DRAM 的制造工艺与目前的 CMOS 逻辑电路的制造工艺不兼容，目前还很难在同一个硅片上制造 CMOS 逻辑电路和 DRAM 存储器^①。这也是为什么片上存储器都采用 SRAM 的一个原因。

① 最新的解决方案包括采用 TSV（过硅通孔，Through Sillicon Vias）技术将 DRAM 芯片进行 3D 堆叠，并形成所谓“混合存储器立方体”（HMC）。有兴趣的读者可以参阅本章扩展阅读。

1. 各种 DRAM 技术

1) 同步 DRAM（SDRAM）

SDRAM 与处理器的数据交换同步于外部的时钟信号，经过初始数据所需要的时延（CAS Latency）后，将以处理器/存储器总线的最高速度运行，而不会插入等待状态。SDRAM 在经过最初的读写延迟后能够以猝发（burst）方式进行访问，每个时钟访问一次，总线速度可达 100MHz 甚至更高。SDRAM 通过内部 interleaving 实现快速访问；SDRAM 内部采用分体（Bank）设计，提供对 SDRAM 的流水访问方式。整个存储器分成容量相同的多个个体，允许访问某个 Bank 的同时，对其他 Bank 进行数据准备。

2) DDR SDRAM

DDR SDRAM 的全称为 Double Data Rate SDRAM，它是在 SDRAM 的基础上改进而来的。SDRAM 在一个时钟周期内只传输一次数据，它只利用了时钟的上升沿进行数据传输；而 DDR SDRAM 则是在一个时钟周期内传输两次数据，它能够在时钟的上升沿和下降沿各传输一次数据，因此被称为双倍速率同步动态随机存储器。图 5-32 中对比了 DDR SDRAM 与 SDRAM 的数据传输特性，理论上在相同的 SDRAM 核心频率和外部时钟频率下，DDR SDRAM 的数据总线带宽理想值是 SDRAM 的两倍。

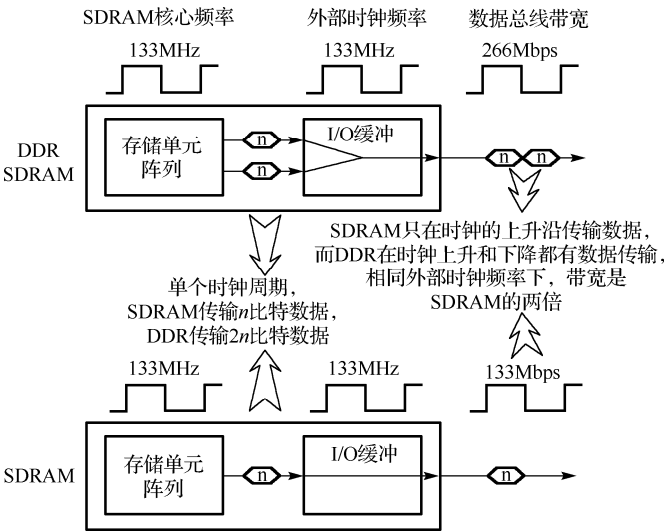


图 5-32 DDR SDRAM 与 SDRAM 数据传输特性对比

表 5-2 详细列举了 SDRAM 和 DDR SDRAM 在命令、功能与结构、封装与电气特性上的不同点。在结构上，DDR SDRAM 具有更先进的同步电路，它不仅采用了反相时钟，而且还使用了 DLL（Delay Locked Loop）来同步命令信号和数据信号，因而它可以支持更高的工作频率；另外，DDR SDRAM 芯片还提供了 DQS 信号接口，专门用于采样数据，增强了数据读写的稳定性。在电气特性上，DDR SDRAM 使用了 2.5V 的 SSTL_2 接口，与 SDRAM 相比其功耗也降低了。

表 5-2 DDR SDRAM 和 SDRAM 的对比

比较项	内存类型	SDRAM	DDR SDRAM
功能与结构			
时钟		单一时钟	反相时钟
工作频率		100MH、133MHz	133MHz、166MHz、200MHz

续表

比较项 \ 内存类型	SDRAM	DDR SDRAM
功能与结构		
预取设计	1 比特	2 比特
数据传输率	1/时钟周期	2/时钟周期
CAS 延迟 (CL)	2、3	2、2.5、3
写入延迟	0	可变
猝发长度 (BL)	1、2、4、8、全页	2、4、8
锁相环	可选	工作时必需
自动刷新间隔周期	固定	弹性设计 (最大值与 SDRAM 的固定值相同)
数据选通信号 (DQS)	无	必需
封装与电气特性		
封装类型	TSOP-II	TSOP-II
工作电压	3.3V (LVTTL)	2.5V (SSTL_2)

3) DDR2 SDRAM

DDR2 SDRAM (Double Data Rate 2 SDRAM, 第二代同步双倍速率动态随机存取存储器), 以下简称为 DDR2。与 DDR 相比, DDR2 有了许多改进, 表 5-3 列出了 DDR2 和 DDR 的主要区别。

表 5-3 DDR2 和 DDR 的主要区别

项目		DDR2 SDRAM	DDR SDRAM
基本设计	时钟频率 (MHz)	200/266/333	100/133/166/200
	数据传输速率 (Mbps)	400/533/667	200/266/333/400
	预取设计	4 比特	2 比特
	猝发长度	4/8	2/4/8
	逻辑块数量	最多 8 个	最多 4 个
	CAS 延迟	3、4、5	1.5、2、2.5、3
	数据选取脉冲	差分数据选取脉冲	单数据选取脉冲
电气性能与封装	工作电压	1.8V	2.5V
	接口标准	SSTL_18	SSTL_2
	功耗	304mW (最大) 533Mbps	418mW (最大) 266Mbps
	封装	CSP (FBGA) 无铅封装, 60/64/68/84/92 引脚	TSOP-II (66 引脚)、 CSP (60 引脚)
	模组标准	240 引脚 DIMM	184 管脚 DIMM
功能	命令集	同 DDR	—
	基本时序定义	同 DDR	—
	新功能	ODT、ODT 调校、Posted CAS、AL	无

在相同核心频率条件下, DDR2 的实际带宽是 DDR 的两倍。这是因为 DDR2 中采用了 4 比特的预取。所以, 虽然 DDR 和 DDR2 一样都是在时钟的上升沿和下降沿触发数据的, 但是 DDR2 的预取能力是 DDR 的两倍, 所以在同样的核心频率的条件下, DDR2 的传输速率就是 DDR 的两倍。举个例子, 当核心工作频率是 50MHz 且存储器的位宽为 8 时, SDRAM 的数据传输能力是 50MB/s, DDR 的数据传输能力是 100MB/s, 而 DDR2 的数据传输能力是 200MB/s。

4) DDR3 SDRAM

DDR3 SDRAM (Double Data Rate 3 SDRAM, 第三代同步双倍速率动态随机存取存储器), 以下简称为 DDR3。

DDR3 提供了相较于 DDR2 SDRAM 更高的运行效率与更低的电压，是 DDR2 SDRAM 的后继者，也是现时流行的内存产品。对比 DDR2，DDR3 采用了一些新的设计：

- 8 比特预取设计，而 DDR2 为 4 比特预取，所以 DRAM 内核的频率只有等效数据频率的 1/8，DDR3-800 的核心工作频率（内核频率）只有 100MHz。
- 采用点对点的拓扑架构，以减轻地址/命令与控制总线的负担。
- 采用 100nm 以下的生产工艺，将工作电压从 1.8V 降至 1.5V，增加异步重置（Reset）与 ZQ 校准功能。

5) DDR4 SDRAM

目前的 DDR3 内存最高标准频率为 2133MHz，电压则有标准版 1.5V、节能版 1.35V 两种。DDR4 将继续沿着高频率、低电压之路前进。

DDR4 内存会带来频率的大幅提升，更会有 1.2V 低电压、更好的对等保护和错误恢复等技术。DDR4 内存将会拥有两种规格。其中使用 Single-endedSignaling 信号的 DDR4 内存其传输速率已经被确认为 1.6~3.2Gbps，而基于差分信号技术的 DDR4 内存其传输速率则可以达到 6.4Gbps。由于通过一个 DRAM 实现两种接口基本上是不可能的，因此 DDR4 内存将会同时存在基于传统 SE 信号和差分信号的两种规格产品。DDR4 内存将会是 Single-endedSignaling（传统 SE 信号）方式和 DifferentialSignaling（差分信号技术）方式并存。

2. SDRAM 结构

SDRAM 存储体是由 DRAM 存储单元构成的，存储单元的结构决定了 SDRAM 的工作原理。图 5-33 为 DRAM 存储单元原理图，DRAM 存储单元由 MOS 管 M 和电容 C_1 组成， C_1 是存储数据用的，M 管是控制数据读出的， C_2 为寄生电容，存储单元的逻辑是 1 还是 0 是由 C_1 的电位决定的。电容的电位为 V_{CC} 时，存储单元的逻辑是 1；电容的电位为 GND 时，存储单元的逻辑是 0。DRAM 存储单元的访问延时和电容值有关，电容值越小，访问 DRAM 所需要的时间越短。

DRAM 内部采用栅格存储信息，按行、列方式进行排列。存储控制器访问时，首先发送“行选”行地址选通信号与“列选”列地址选通信号，选中相应的地址单元后，数据才可以进行传输。

DRAM 存储单元读/写操作的具体原理是：在读写数据的时候，存储单元被字线和位线激活，字线控制 MOS 管 M 打开，电容 C_1 的电荷由位线读出或写入，敏感放大器（Sense Amplifier）根据位线和参考线之间差分电压来决定 C_1 中存储的逻辑值。因为在 MOS 管 M 导通后，电容 C_1 里的电荷及位线上的寄生电容 C_2 共同形成了一个新电位，再加上漏电流的存在，位线上的电平值会和这之前电容 C_1 的电平值有很大误差，所以在读取数据前，参考线上的电压稳定为电容最高电压的一半（ $1/2 V_{CC}$ ），若参考线电压高于位线电压，就判断为逻辑 0，否则就判断为逻辑 1，就此准确判断出电容 C_1 存储的逻辑值。若对单元进行写操作，则将数据放到位线上后，当位线上的电压达到 V_{CC} 或 GND 时，感应放大器将 1 或 0 写入单元中。

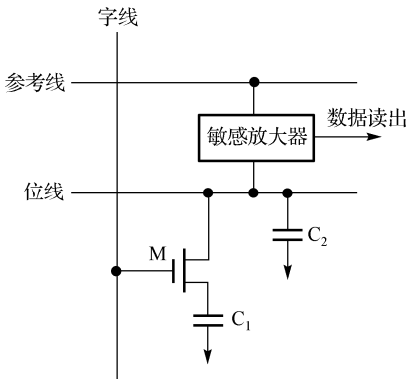


图 5-33 DRAM 存储单元结构图

图 5-34 是 SDRAM 芯片的存储结构图，其中，列地址译码出来的线即对应于图 5-33 中的字线，灰色小方块对应于一个字的数据（N 位数据，N 是 SDRAM 芯片位宽）。SDRAM 采用行、列地址分离的技术（也就是访问 SDRAM 时，SDRAM 控制器首先在地址总线上输出行地址，然后再输出列地址，

这与 SRAM 读写时将地址一次性输出到地址总线是不同的), 行、列地址复用一条总线, SDRAM 控制器 (一般而言, SDRAM 控制器是 EMI 的一部分) 使用 RAS 行选信号和 CAS 列选信号来标识在地址线上传输的是行地址还是列地址。当行地址有效时, SDRAM 存储器的存储阵列中的一行数据被读出到敏感放大器中 (这个过程也称为行激活); 当列地址有效后, 便可以在敏感放大器中读出选中单元的数据。

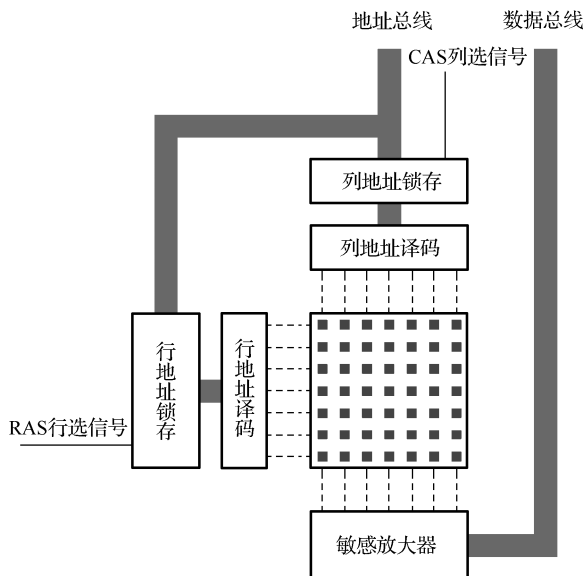


图 5-34 一个 SDRAM 存储体 (Bank) 的结构图

由于 SDRAM 的敏感放大器具有独占性, 所以在进行完一行数据的读写操作后, 必须先关闭该行, 然后才能激活 (Activate) 下一行数据。行的关闭是通过预充电 (Precharge) 命令来实现的, 预充电将当前激活行 (也就是处于敏感放大器中的行) 的数据重写回存储体, 并对行地址进行复位, 同时释放敏感放大器, 为新的一行的激活做好准备。具体而言, 就是将敏感放大器中的数据回写, 即使是没有工作过的存储体也会因行选通而使存储电容受到干扰, 所以也需要敏感放大器进行重写操作。

另外, SDRAM 存储体是采用 DRAM 存储单元里的电容来存储数据的, 由于漏电流的存在, 数据不能长时间地保存在存储单元中, 需要周期性刷新, 所以在 SDRAM 存储器工作时, 必须定期对其进行刷新 (Refresh) 操作。刷新操作与写入操作很相似, 先打开字线, 然后敏感放大器读出每个存储单元的数据再重新写入。

完整的 SDRAM 芯片结构如图 5-35 所示, 它可以看成由体 (Bank)、行 (Row)、列 (Column) 构成的一个三维结构。每个 Bank 是一个独立的存储区域, 拥有自己独立的敏感放大器。现代 SDRAM 存储器通常在一颗存储器芯片中设计 4 个甚至更多 Bank, 这也就意味着一颗存储器芯片可以同时激活 4 个或更多个行 (因为每个 Bank 都有自己的敏感放大器)。一行对应于敏感放大器的数据宽度, 不同容量的 SDRAM 芯片可以有不同的行大小 (也就是一行数据的容量)。有时我们也将一行数据称为一页数据 (Page)。在 SDRAM 的读时序中, 被访问的行首先被激活到敏感放大器中, 然后再根据列地址选择输出相应列的数据。比如, 某款 SDRAM 芯片内部分为 4 个 Bank, 每个 Bank 又分为 4096 个行, 每行包含 256 个单元, 每个单元为 16 位, 即 $4\text{Banks} \times 4096\text{行} \times 256\text{单元} \times 16\text{位}$ 。所以, 每个 Bank 为 2MB, 整个芯片为 8MB。因为每个单元是 16 位, 所以需要两个芯片才能构成 32 位系统。一般来说, 硬件系统中采用并联两片这种 SDRAM 芯片的连接方式, 其中一片存放高 16 位数据, 另一片存放低 16 位数据。

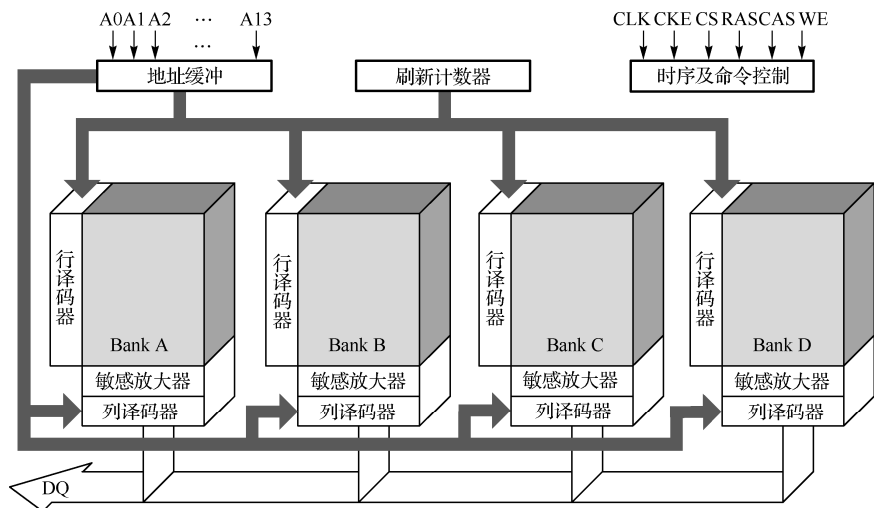


图 5-35 SDRAM 芯片结构图

一个完整的 SDRAM 芯片访问分为 3 个阶段：行选周期、列选周期、预充电周期。每个 Bank 都有一个行缓冲区（Row Buffer，也就是敏感放大器），行缓冲区中的数据构成了一个 Page。当一个 Bank 接收到行地址时（对应于激活命令），它进入行选周期，锁存行地址并将这一行读出放大到行缓冲区中；当 Bank 接收到列地址时（对应于读写命令），它进入列选周期，根据送来的命令进行相应的读写操作；当需要关闭该行时，即发出预充电命令，SDRAM 进入预充电周期，它负责将行缓冲区中的数据写回 SDRAM 的存储体中。若下一次读写操作的目的地址正好位于激活的行（也就是该地址对应的行当前已被缓冲在敏感放大器中，称为行缓冲区命中或行命中），则可以直接发送列选命令就能读写数据，操作可以流水执行。而当需要读写的数据不在行缓冲区时，就需要将原来打开的行关闭，然后激活需要操作的行，由于这些操作只能顺序执行，所以读写延迟很大。因为 SDRAM 芯片是由多个 Bank 组成的，多个 Bank 可以同时处于行选周期，整个 SDRAM 芯片就拥有多个行缓冲区，读写操作的性能相比单个 Bank 要好得多。

此外，在 SDRAM 芯片中有刷新计数器（Refresh Counter），它会自动累加当前的刷新地址，对 SDRAM 进行刷新操作时不需要提供操作地址，只要由 SoC 芯片的 SDRAM 控制器定时向 SDRAM 芯片发送刷新命令即可，这种是一般情况下使用的刷新方式，称为自动刷新（Auto-Refresh）。另外，SDRAM 芯片还提供自刷新（Self-Refresh）机制，在自刷新模式下，SDRAM 芯片处于低功耗模式，此时只需要很小的电流来维持刷新操作，刷新操作完全由 SDRAM 内部电路完成，不需要 SDRAM 控制器发出刷新命令，这使得 SoC 芯片在进入低功耗休眠模式（此时 SDRAM 控制器也会进入休眠模式而无法继续发出定时的刷新命令）时 SDRAM 中存放的数据不会丢失。

本质上，SDRAM 芯片并不能称为随机访问设备，因为它的三维组织结构使得内部不同存储单元的访问时间也不同。这主要是由 Bank 中页的预充电（Precharge）和激活（Activation）延迟造成的。

3. 访问 SDRAM

SDRAM 的读写操作分为：初始化、行有效、列读写、猝发操作。

1) 初始化

- ① 激活命令（Active），用来打开某个 Bank 内的一行，由地址线 A0~A11 择行，当选择同一 Bank 内不同的行时，需要首先用预充电（Precharge）命令将当前激活行写回，然后才能使用激活命令将新的行激活到敏感放大器；
- ② 预充电（Precharge）命令用来关闭指定 Bank 的打开行，A10 决定一个还是所有 Bank 都预充

电（在行地址有效时，A10 是行地址的一根线，列地址的范围用不到 A10，所以可用 A10 来决定一个还是所有 Bank 都预充电），当只有一个 Bank 需要预充电时，用 BA0、BA1 选择 Bank，预充电以后处于空闲状态，在经过预充电命令 t_{RP} （行预充电时间）时间后，才可以执行下一次行激活命令。

2) 行有效

初始化完成后，要想对一个 Bank 中的阵列进行寻址，首先就要确定行（Row），使之处于活动状态（然后再确定列）。虽然之前要进行片选和 Bank 的定址，但它们与行有效可以同时进行。图 5-36 中即是行有效的时序图。RAS（Row Address Strobe）表示行寻址。

从图 5-36 中可以看出，在 CS_n 、Bank 定址的同时，RAS（Row Address Strobe，行地址选通脉冲）也处于有效状态。此时地址线则发送具体的行地址。大多数 SDRAM 共有 12 个地址线，由于是二进制表示法，所以共有 4096 行，A0~A11 的不同数值就确定了具体的行地址。由于行有效的同时也是相应 Bank 有效，所以行有效也可称为 Bank 有效。

3) 列读写

行地址确定之后，就要对列地址进行寻址了。行地址与列地址线是共用的。列寻址信号与读写命令是同时发出的。但没有一个信号是发送读或写的明确命令的，而是通过芯片的可写状态的控制来达到读/写的目的。CAS（Column Address Strobe，列地址选通脉冲）信号可以区分行与列寻址的不同，配合 A0~A9 和 A11 来确定具体的列地址。图 5-37 为列有效时序图。

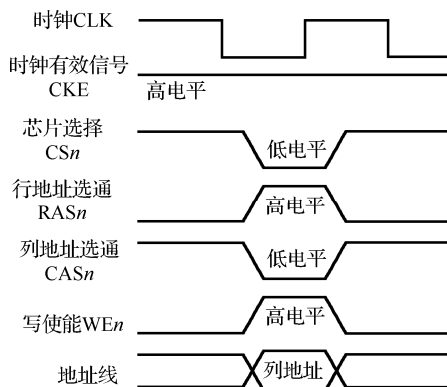


图 5-36 行有效时序图

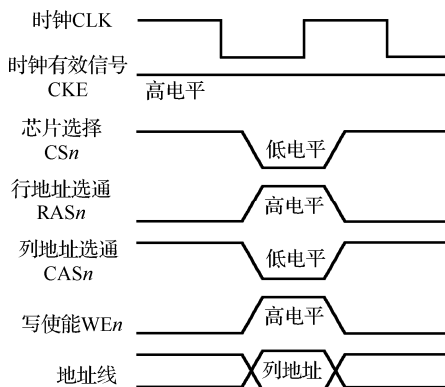


图 5-37 列有效时序图

然而，在发送列读写命令时必须要与行有效命令有一个间隔，这个间隔定义为 t_{RCD} ，即 RAS to CAS Delay（RAS 至的 CAS 的延迟），也可以理解为行选通周期，这是根据芯片存储阵列电子元件响应时间（从一种状态到另一种状态变化的过程）所指定的延迟。 t_{RCD} 是 SDRAM 的一个重要时序参数，编写 SDRAM 控制器初始化代码时，我们往往以时钟周期（ t_{CK} ，Clock Time）数作为 t_{RCD} 的单位，比如 $t_{RCD}=2$ ，就代表延迟周期为两个时钟周期^①。图 5-38 为 t_{RCD} 时序示意图。

数据写入与输出在选定列地址后就已经确定了具体的存储单元，剩下的事情就是数据通过数据 I/O 通道（DQ）输出到内存总线上了。但是在 CAS 发出之后，仍要经过一定的时间才能有数据输出，从 CAS 与读取命令发出到第一个数据输出的这段时间被定义为 CL（CAS Latency，CAS 延时）。配置 SDRAM 控制器时，CL 的单位与 t_{RCD} 一样，为时钟周期数^②。在 CAS 发出后，如果是读数据，则需要再等待 CL 的时间才能从数据线上得到数据，如图 5-39 所示。

① t_{RCD} 时间的长短取决于存储器芯片的设计与制造工艺，比如对于 Micron 公司 2Gb DDR3-SDRAM 芯片而言，这个时间大概是 13~15ns，对于 400MHz 主频的芯片而言，这大概是 5~6 个时钟周期。

② Micron 公司 2Gb DDR3-SDRAM 芯片的 CL 时间也是 13~15ns，对于 400MHz 主频的芯片而言，这大概是 5~6 个时钟周期。

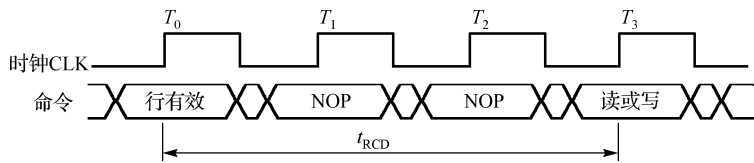


图 5-38 t_{RCD} 时序示意图

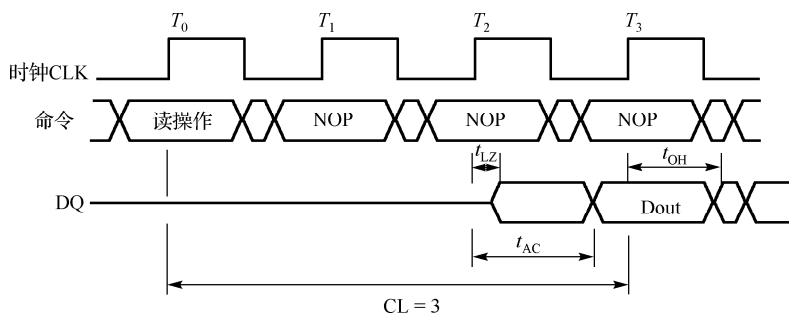


图 5-39 CAS 潜伏期时序示意图

3) 猝发操作

猝发（Burst）是指在同一行中相邻的存储单元连续进行数据传输的方式，连续传输所涉及存储单元（列）的数量就是猝发长度（BL，Burst Lengths）。

上文介绍的读/写操作都是一次对一个存储单元进行寻址，如果要连续读/写则还要对当前存储单元的下一个单元进行寻址，也就是要不断地发送列地址与读/写命令（行地址不变，所以不用再对行寻址）。虽然由于读/写延迟相同可以让数据的传输在 I/O 端是连续的，但它占用了大量的存储控制资源，在数据进行连续传输时无法输入新的命令，效率很低。为此人们开发了猝发传输技术，只要指定起始列地址与猝发长度，内存就会依次地自动对后面相应数量的存储单元进行读/写操作而不再需要存储控制器连续地提供列地址。这样，除了第一个数据的传输需要若干个周期（主要是之前的延迟，一般的是 $t_{RCD}+CL$ ）外，其后每个数据只需一个周期即可获得。

如图 5-40 和图 5-41 所示，两种方式的区别如下：

- 非猝发连续读取模式：不采用猝发传输而是依次单独寻址，此时可等效于 BL=1。虽然可以让数据连续传输，但每次都要发送列地址与命令信息，控制资源占用极大。^①
- 猝发连续读取模式：只要指定起始列地址与猝发长度，寻址与数据的读取自动进行，而只要控制好两段猝发读取命令的间隔周期（与 BL 相同）即可做到连续的猝发传输。

SDRAM 芯片进行页访问时，SoC EMI（SDRAM 控制器）必须根据当前操作地址对应 Bank 的状态来发出不同的命令：

- <1> 如果该 Bank 中没有页处于激活状态，则
 - <1.1> 激活当前操作地址对应的页；
 - <1.2> 对当前操作地址对应的列进行页读写操作；
- <2> 如果该 Bank 中有页处于激活状态（每个 bank 只可能有一个页激活），则
 - <2.1> 如果已经激活的页就是当前操作地址对应的页，则，
 - <2.1.1> 对当前操作地址对应的列进行页读写操作；

① SEP4020 处理器的 SDRAM 控制器所采用的就是这种非猝发连续读取模式，由于每次只发出一拍传输，SDRAM 控制器不会发出 terminate 命令。在 AHB 的 incr 模式下，SDRAM 控制器会预取下一个数据。如果 incr 传输停止，或者预取到了行边界，预取停止。

<2.2> 如果已经激活的页不是当前操作地址对应的页，则

<2.2.1> 对该 Bank 进行预充电，将已经激活的页关闭；

<2.2.2> 激活当前操作地址对应的页；

<2.2.3> 对当前操作地址对应的列进行页读写操作；

由此可见，SDRAM 的这种页组织特性使得访问不同页的存储单元时，由于需要进行预充电（Precharge）和激活（Activation）的换页操作，造成了存储单元访问时间不同，这个过程带来的延时可能是读取时间的几倍至几十倍。

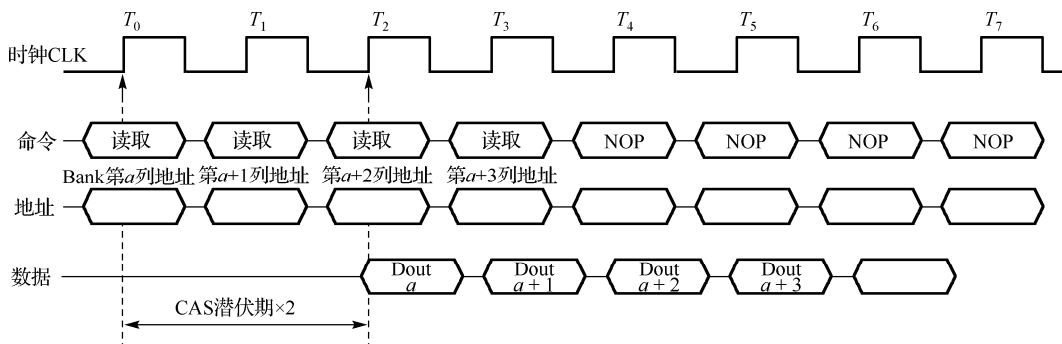


图 5-40 非猝发连续读取模式时序示意图

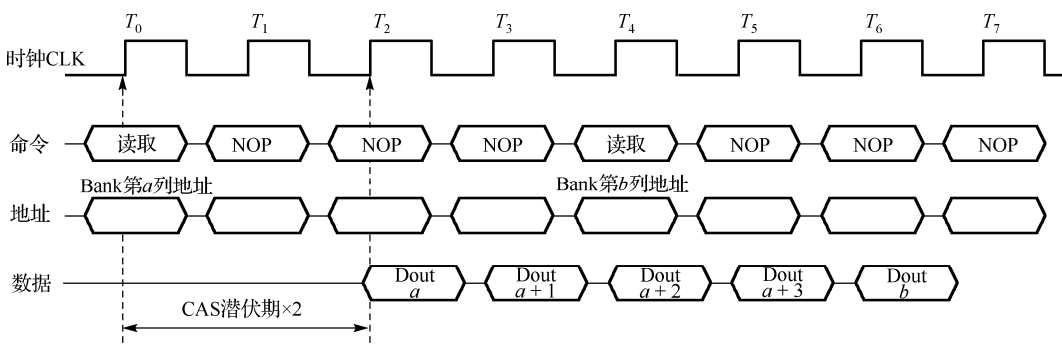


图 5-41 猝发连续读取模式时序示意图

5) 开页策略与关页策略

SDRAM 读写过程中，首先要将被操作的行（有时也称为“页”，page）激活到敏感放大器，才能进行读写。在激活一个新行前，还必须将当前在敏感放大器中的行预充电回存储体。不管是 ACTIVE 命令还是 PRECHARGE 命令都需要一定的时间，由于 SDRAM 读写机制的特点，存在两种不同的管理策略：所谓开页（Open Page）策略和关页（Close Page）策略。在开页策略中，被激活的行将一直保存在敏感放大器中，直到处理器需要访问的内容在另外一个新行中。此时 SDRAM 控制器将发起一次 PRECHARGE 命令，将当前行关闭，然后控制器将发起一次 ACTIVE 命令，将新的目标行激活。而在关页策略中，每次完成对当前行的访问后，SDRAM 控制器将发起一次 PRECHARGE 命令，将当前行关闭。处理器的下一次访存请求将首先执行 ACTIVE 命令。若连续访问不同行，下一个访存命令到达时，虽然与上一个命令有 Bank 冲突的发生，但是由于敏感放大器已经在上次命令结束后进行预充电操作了，因此本次访存不再需要进行预充电操作，而直接进行行激活与列访问操作即可。在这种情况下，访存延时较使用开页策略的时候小。显然，对于局部性较好的访存操作，开页策略将比较有

利于降低访存延迟；而对于访存局部性不好的情况，关页策略比较有利。程序员可以通过配置 SDRAM 控制器中相应的寄存器选择采用开页策略还是关页策略。

4. DDR SDRAM 的主要命令

SDRAM 芯片是通过接收命令来进行一系列操作的。表 5-4 为 DDR SDRAM 芯片的常用命令缩写及其说明，与 SDRAM 芯片的命令基本相同。送给 DDR SDRAM 的命令是由各个控制信号引脚的不同高低电平来决定的，表 5-5 为 DDR SDRAM 指令对应的控制信号真值表。也就是说，SDRAM 控制器发送给存储器的命令并不是通过专门的命令通道传输的，SDRAM 控制器通过在某一个时刻在这些控制信号引脚上输出不同的高低电平来发送相应的控制命令。这个过程非常复杂，好在作为系统设计人员（软件程序员）不需要知道其中的细节，SDRAM 控制器会自动完成这个复杂的控制过程。

表 5-4 DDR SDRAM 常用命令

命令	符号	操作说明
Device deselect	DESL	不对当前设备进行操作
No operation	NOP	空操作
Burst stop	BST	终止猝发读或写
Read	READ	开始猝发读
Read with auto precharge	READA	开始猝发读，当猝发读结束后，自动预充电
Write	WRIT	开始猝发写
Write with auto precharge	WRITEA	开始猝发写，当猝发写结束后，自动预充电
Bank activate	ACT	行激活操作
Precharge	PRE	对选中的 Bank 进行预充电
Precharge all banks	PALL	对所有的 Bank 进行预充电
Mode register set	MRS	设置模式寄存器
Extend mode register set	EMRS	设置扩展模式寄存器，用来配置是否使能 DLL
Auto refresh	REF	自动刷新
Self refresh entry	SELF	进入自刷新模式
Self refresh exit	SREX	退出自刷新模式
Power down entry	PWDN	进入低功耗模式
Power down exit	PDEX	退出低功耗模式

- (1) No operation: 在没有其他 SDRAM 命令的时候，通常会使用 NOP 指令，避免 SDRAM 被上一个指令的信号干扰。此外在猝发传输模式时，也会插入这个命令以避免猝发传输被其他指令影响。
- (2) Mode register set: 这个命令是将设定的配置载入到 SDRAM 中的模式控制寄存器中。主要负责设定猝发传输长度、猝发传输类型以及 CAS 延迟等参数。
- (3) Extend mode register set: 这个命令是将设定的配置载入到 DDR SDRAM 中的扩展模式控制寄存器中。
- (4) Bank activate: 在 Read/Write 操作之前，Activate 命令用来将位于存储器阵列中某一个行的内容放到行缓冲器中以供存取。执行 Activate 命令时需要指定 Bank 和行地址。而当我们激活同一个 Bank 的另一个行时，必须要先将目前位于行缓冲器的信息写回存储器阵列中。
- (5) Read/Write: 利用 Activate 命令把指定的行置入行缓冲器之后，使用 Read 命令将行缓冲器中的信息读出，再经过 CAS latency 的延迟时间，数据便会出现在 DQ 总线上，而执行 Write 命令则是将 DQ 总线上的信息配合 DQM，写入到行缓冲器中。在执行 Read/Write 时需要指定列地址，Write 命令还需将要写入的信息放在 DQ 总线上准备写入。当进行一连串的猝发传输时，我们只需在读写第一个

数据时发出 Read/Write 命令，之后 SDRAM 便会依据模式控制寄存器中所设定的猝发长度做指定长度的猝发传输。

(6) Precharge/Precharge all banks: 相对于 Activate 是用来激活某行的，Precharge 则用来关掉某行。Precharge 命令用来将某一个 Bank 或全部的 Bank 里行缓冲器的信息写回到存储器阵列中。执行 Precharge 命令时需要指定 Bank 地址或是利用 A10 指定 Precharge 所有 Bank。

(7) Auto refresh: 用来维持位于 SDRAM 存储器阵列中的信息。SDRAM 存储信息是以电容的方式，使用 SDRAM 时，需要周期性地刷新位于存储器阵列中的信息。每执行一次 Auto refresh 命令就会对存储器阵列进行刷新，Auto refresh 的行地址由 SDRAM 的内部自行产生，不需要由 SDRAM 控制器给出地址，在每一次执行 Auto refresh 命令后，内部的行地址计数器会自己累加。

(8) Self refresh: Self refresh 命令使得 SDRAM 进入 Self-Refresh 模式，来维持省电模式下 SDRAM 存储器阵列中的信息。在 Self-Refresh 状态时，SDRAM 不接收外部的时钟信号，而是使用内部时钟进行刷新。此时，除了 CKE 必须维持在低电平以外，其他的输入引脚都不会对 SDRAM 造成影响。而要跳出 Self-Refresh 模式需要执行一连串的命令，才能回到正常工作模式，所以会影响到系统的性能。

(9) Power down: 通过 CKE 置于低电平并且伴随着无任何操作的 No operation，SDRAM 进入 Power-Down 模式，此时除了 CKE 外的所有输入/输出均无效。

表 5-5 DDR SDRAM 指令真值表

命令	CKE		CSn	RASn	CASn	WEn	ADDRESS			
	n-1	n					BA0	BA1	A10	A0~A9,A11
不选择	H	X	H	X	X	X	X		X	X
无操作	H	X	L	H	H	H	X		X	X
读终止	H	X	L	H	H	L	X		X	X
猝发读操作	H	X	L	H	L	H	Valid		L	Valid
带自动预充电的读									H	
猝发写操作	H	X	L	H	L	L	Valid		L	Valid
带自动预充电的写									H	
Bank 激活	H	X	L	L	H	H	Valid		Valid	Valid
预充电	H	X	L	L	H	L	Valid		L	X
全部预充电							X		H	X
设置模式寄存器	H	X	L	L	L	L	L	L	L	Valid
设置扩展模式寄存器							H	L	L	Valid
自动刷新	H	H	L	L	L	H	X		X	X
进入自刷新	H	L								
退出自刷新	L	H	H	X	X	X	X		X	X
进入低功耗	H	L	H	X	X	X	X		X	X
			L	H	H					
退出低功耗	L	H	H	X	X	X	X		X	X
			L	H	H					

5. DDR SDRAM 操作时序

1) DDR SDRAM 读操作

DDR SDRAM 的读操作过程与 SDRAM 非常类似（毕竟 DDR 技术是对 SDRAM 的升级）。读操作涉及激活 Bank 和发送读命令，对一个 Bank 的指定行列进行读写时，需要按照下面的步骤来

执行，如图 5-42 所示。图中给出了 CL 延时为 2 个时钟周期的连续 4 拍（也就是猝发长度为 4）猝发传输。

- （1）激活 Bank，发送 ACT (Activate) 命令，同时送出行地址和 Bank 地址，激活所选的 Bank 并打开指定行。
- （2）发送 READ 命令同时送出列地址和 Bank 地址，该命令至少在发送 ACT 命令 t_{RCD} 后发送；发送 READ 命令后等待 CL (CAS Latency)，指定列上的数据出现在数据总线 DQ 上，送出数据的个数由 BL (Burst Length) 决定。
- （3）ACT 命令发送后，至少等待 t_{RAS} 才可发送 PRE (Precharge) 命令。发送 PRE 命令时，若地址线 A10（地址线第 11 位）为低电平，则 Bank 地址决定需要预充电的 Bank；反之，则刷新所有的 Bank。
- （4）发送 PRE 命令 t_{RP} 时间后，选中的 Bank 回到 IDLE 状态，并可响应新的 ACT 命令。

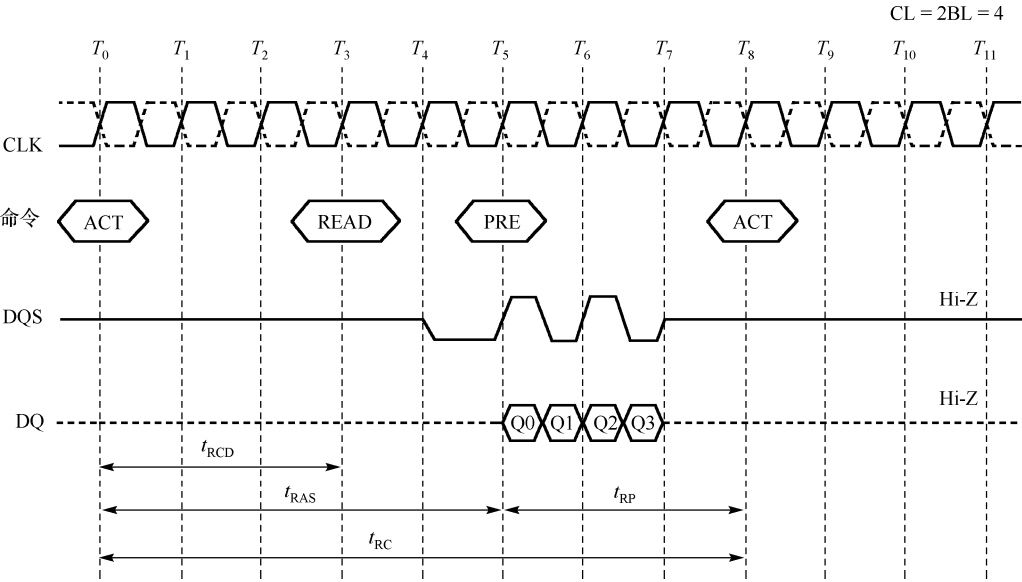


图 5-42 SDRAM 读操作时序图

3) DDR SDRAM 写操作

和读操作一样，写操作前也必须先用 ACT 命令打开一个 Bank 的指定行，如图 5-43 所示，写操作需要遵循以下步骤：

- （1）激活一个 Bank，发送 ACT 命令，同时送出 Bank 地址和行地址，激活所选的 Bank 并打开指定行。
- （2）等待 t_{RCD} 后，发送 WRIT 命令并同时送出起始的列地址，从 WRIT 命令发出的 CLK 上升沿到其后的 DQS 上升沿必须满足最小写入延迟 t_{DQSS} 。
- （3）在 Burst 写操作中，DQS 信号必须与 DQ 信号同步，DQS 信号第一个上升沿必须对准第一个 DQ 数据的中间，以保证所有的数据能在 DQS 的跳变沿被正确捕获。
- （4）在 ACT 命令后至少等待 t_{RAS} 才能进行 Precharge 操作。
- （5）在 Precharge 操作 t_{RP} 后，选中的 Bank 又回到 IDLE 状态，可以响应新的 ACT 命令。

3) DDR SDRAM 刷新操作

图 5-44 是 SDRAM 刷新操作的时序图，刷新操作必须遵循下面的规范：

(1) 刷新操作必须在所有的 Bank 处于 IDLE 状态时进行。若 SDRAM 存储体不处于 IDLE 状态, 必须将所有的活动 Bank 进行 Precharge, 通过 PALL (发送 PRE 命令的同时将 A10 拉高) 命令来完成。

(2) PALL 后等待 t_{RP} 时间, 所有 Bank 处于 IDLE 状态, 这时可以进行 Refresh 操作, 发送 REF 命令后, SDRAM 存储体内部就会自动刷新内部的一行数据。SDRAM 内部具有行地址生成器 (也称刷新计数器), 用来自动依次生成行地址。由于刷新是针对一行中的所有存储体进行的, 所以无需列寻址。

(3) 发送 REF 命令后, 存储体内部立即进行刷新操作, 等待 t_{RFC} 后刷新操作完成, SDRAM 存储体回到 IDLE 状态, 才能发送下一个 REF 命令, t_{RFC} 是两个 REF 命令之间最小的时间间隔。

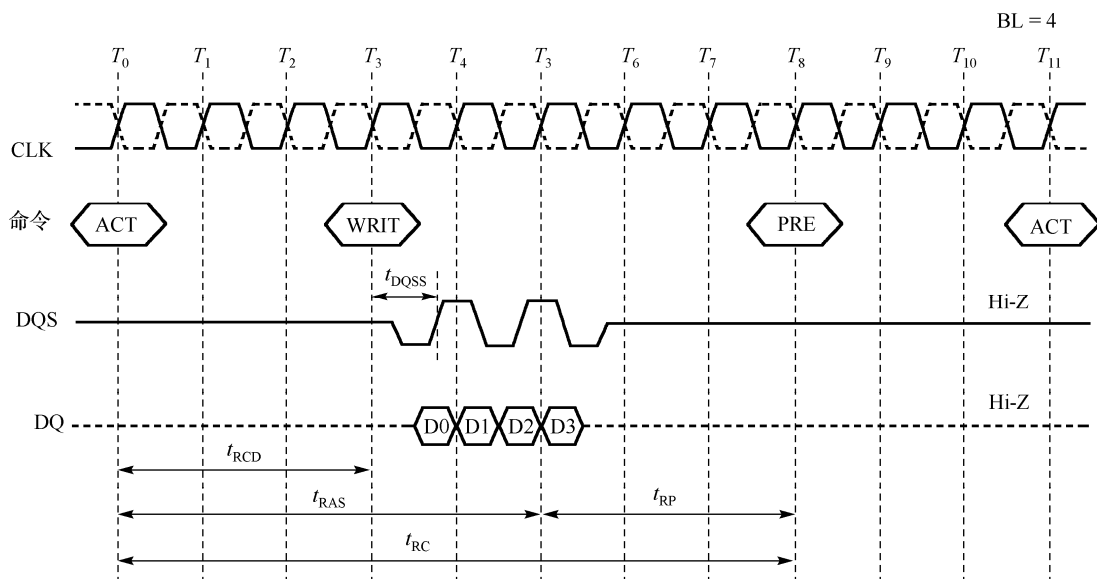


图 5-43 DDR 写操作时序图

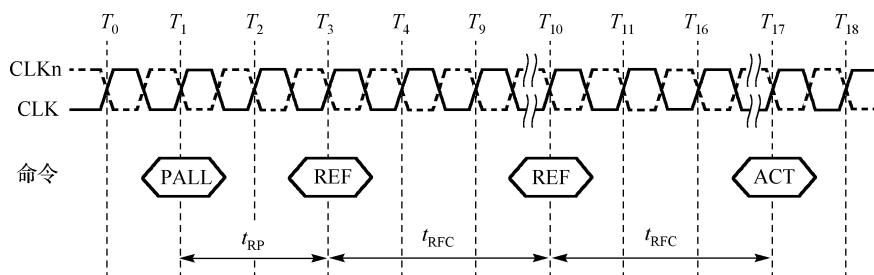


图 5-44 SDRAM 刷新时序图

4) DDR SDRAM 初始化时序

SDRAM 在上电之后是没有确定的逻辑状态的, 在进行读写操作之前, 必须经过一个规范的上电过程并符合规定的初始化时序。

上电的具体过程如图 5-45 所示, 首先拉高 VDD, 然后拉高 VDDQ, 而后将 VREF 和 VTT 置为高电平; 将 CKE 保持在低电平以确保 DQ 和 DQS 处于高阻状态; 在所有的电压和时钟稳定之后, 等待 200 μ s, 最后将 CKE 拉高并发送 DSEL 或 NOP 命令。

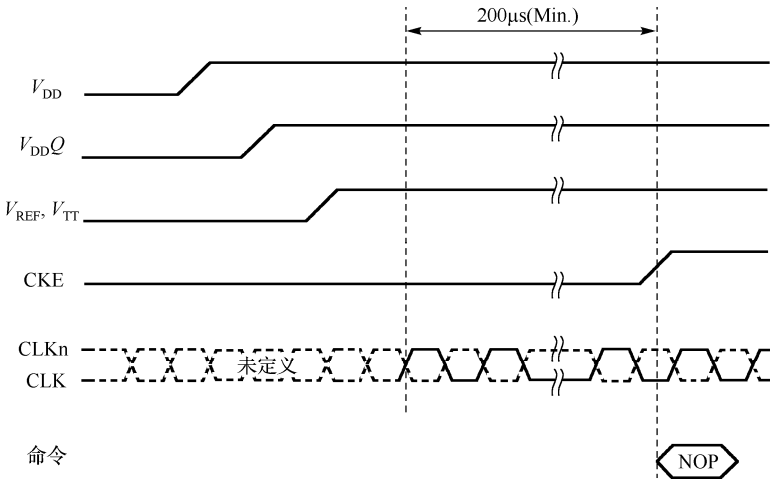


图 5-45 SDRAM 上电操作

如图 5-46 所示，上电操作完成后，还要执行一连串的命令：

- 对所有 Bank 进行 Precharge 操作（发送 PALL 命令），让所有 Bank 处于 IDLE 状态。
- 发送扩展模式寄存器设置命令（EMRS）使能 DLL。
- 通过模式寄存器设置命令（MRS）重置 DLL 并配置 SDRAM 的模式参数。
- 发送 PALL 命令 Precharge（预充电）所有 Bank，然后进行至少两次刷新操作（REF）。
- 通过模式寄存器设置命令（MRS）重新配置 SDRAM 的模式参数。
- MRS 操作过后等待 t_{MRD} 就能对 SDRAM 发送除 READ 外的其他命令，直到 EMRS 命令发送 200 个周期后，才能对 SDRAM 发送 READ 命令。

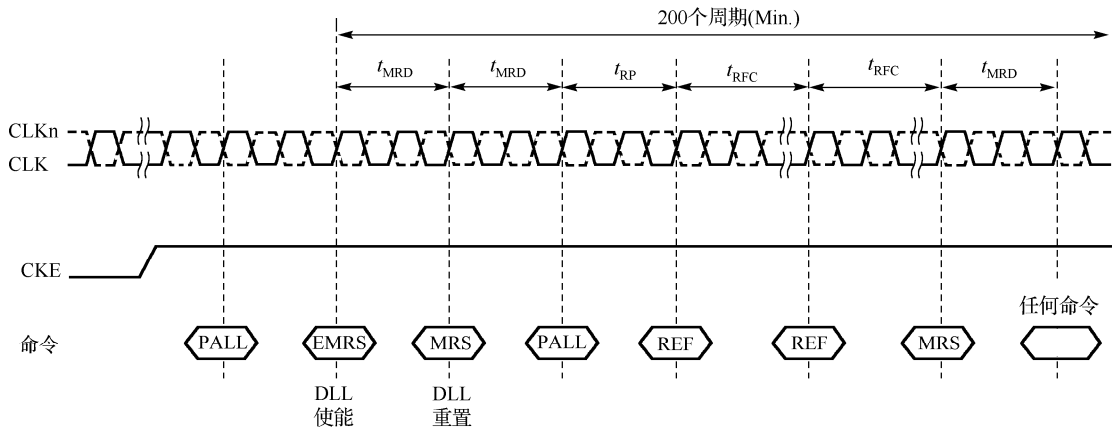


图 5-46 DDR SDRAM 初始化时序图

5.4.3 非易失性存储器

嵌入式系统必须使用非易失性存储器，非易失性存储器可以在系统掉电情况下保存数据。非易失性存储器有多种实现方式，容量、尺寸、性能及可靠性各不相同。目前嵌入式系统中常用的非易失性存储器为固态存储器（solid state memory）。当前主流固态存储器为 Flash 存储器。Flash 存储器又称闪存，全名叫 Flash EEPROM（Electrically Erasable Programmable Read-Only Memory，电可擦除可编程只读存储器）。

最早的非易失性存储器为 ROM (Read-Only Memory, 只读存储器)。ROM 内部的资料是在 ROM 的制造工序中在工厂里用特殊的方法烧录进去的, 其中的内容只能读不能改, 一旦烧录进去, 用户只能验证写入的数据是否正确, 不能再做任何修改。如果用户需要更新数据, 只能舍弃不用, 重新订做一份。ROM 是在生产线上生产的, 由于成本高, 一般只用在大批量应用的场合。

由于 ROM 制造和升级的不便, 后来人们发明了 PROM (Programmable ROM, 可编程 ROM)。最初从工厂中制作完成的 PROM 内部并没有数据, 用户可以用专用的编程器将自己的数据写入, 但是这种机会只有一次, 一旦写入后也无法修改, 若是出了错误, 已写入的芯片只能报废。PROM 的特性和 ROM 相同, 但是其成本比 ROM 高, 而且写入数据的速度比 ROM 的量产速度要慢, 一般只适用于少量有需求的场合或是 ROM 量产前的验证。

EPROM (Erasable Programmable ROM, 可擦除可编程 ROM) 可重复擦除和写入, 解决了 PROM 芯片只能写入一次的弊端。EPROM 芯片有一个很明显的特征, 在其正面的陶瓷封装上开有一个玻璃窗口, 透过该窗口可以看到其内部的集成电路, 紫外线透过该孔照射内部芯片就可以擦除其内的数据, 完成芯片擦除的操作要用到 EPROM 擦除器。EPROM 内资料的写入要用专用的编程器, 并且往芯片中写内容时必须加一定的编程电压 ($V_{PP}=12\sim 24V$, 根据不同的芯片型号而定)。EPROM 芯片在写入数据后, 还要以不透光的贴纸或胶布把窗口封住, 以免受到周围的紫外线照射而使数据受损。

由于 EPROM 操作的不便, EEPROM (Electrically Erasable Programmable ROM, 电可擦除可编程 ROM) 被研制出来并得到广泛使用。EEPROM 的擦除不需要借助于其他设备, 它是以电子信号来修改内容的, 而且不必将数据全部擦除才能写入, 彻底摆脱了 EPROM 擦除器和编程器的束缚。EEPROM 在写入数据时仍要利用一定的编程电压, 属于双电压芯片。此时, 只需用厂商提供的专用刷新程序就可以轻而易举地改写内容。

EEPROM 的一种特殊形式即 Flash 存储器。Flash 存储器属于真正的单电压芯片, 读/写操作都在单电压下进行, 只需利用专用程序即可方便地修改其内容。Flash 存储器的使用类似于 EEPROM, 擦除 Flash 存储器时, 也要执行专用的刷新程序。不同之处在于, Flash 存储器并非以字节为基本单位, 而是以扇区 Sector (或称数据块 Block) 为最小单位, 扇区大小随厂商不同而有所区别; 只有在读数据时, 才以字节为最小单位读数据。Flash 以扇区为单位对整块数据进行擦除, 降低了设计的复杂性, 相比 EEPROM, 可以做到高集成度、大容量, 存储容量普遍大于 EEPROM, 价格也比较合适。此外, Flash 的实现工艺与 EEPROM 也有所不同, 写入速度更快。

Flash 存储器的读速度较快, 由于写数据之前需要先进行块擦除, 然后以整块为单位写回数据, Flash 存储的写速度较读速度要慢得多。目前的 Flash 存储器主要存在两种形式: NOR Flash 及 NAND Flash。这两种形式的 Flash 存储器是以构成存储器单元的典型逻辑门类型命名的。NOR Flash 与 NAND Flash 的关键不同在于 NAND Flash 的容量密度要远高于 NOR Flash。NOR Flash 的容量通常为 MB 级别, 而 NAND Flash 的容量通常为 GB 级别。大部分固态存储器, 诸如 U 盘、MMC、SD 卡, 均采用 NAND Flash, 同时搭配相应控制器提供访问接口。在嵌入式系统中, Flash 存储器通常焊接在电路板上, 通过相应的接口与 SoC 相连。

1. NOR Flash

NOR Flash 的存储空间由数个 Bank 组成, 每个 Bank 包含数个 Sector (扇区)。Sector 可以被独立擦除。用户在读取某一 Bank 的同时, 可以对另一 Bank 进行擦除或编程操作。

NOR Flash 上电后将进入读模式, 在读模式下, 存储器可以进行随机访问, 通过发起读传输, 用户可以读取任意地址空间的数据内容 (NOR Flash 的读时序与 SRAM 的读时序比较类似)。对 NOR Flash 进行擦除或写操作, 需要按规范向 NOR Flash 的特定命令寄存器写入相应命令。对 NOR Flash 进行写

操作之前首先要对数据所在 Sector 或是对整个存储器进行擦除操作。进行擦除操作后，相应 Sector 的内容将全部变为逻辑 1。随后用户可以字节或是字（指 Flash 存储器数据位宽，多为 16 位）为单位对存储器进行编程，写入数据。

对于 NOR Flash 的操作也需要相应的控制器支持。SoC 中的 EMI 通常实现了该控制器的功能。大多数 NOR Flash 提供通用闪存接口（Common Flash Interface）。通过该接口，控制器可以检测存储器的型号、容量及性能信息。闪存的工业标准规范 JEDEC 137-A 以及 JESD68.01 也包括对于该接口的定义。

大多数 Flash 存储器设计有启动扇区（boot sector），对于该区域的写保护策略比其他区域更为严格，健壮性更高。通常，对于该区域的擦写需要设置存储器指定管脚特定的逻辑电平。启动扇区包含系统启动所需的代码，处理器首先从该区域取指执行，然后跳转执行存储在其他区域（或是其他存储器）中的代码，继续启动过程。此外，启动扇区通常还写有恢复（recovery）代码，借助恢复代码，用户可以重新擦写启动扇区之外的存储空间。

上文说到 Flash 存储器通常表贴（SMT）在电路板上，对于 Flash 存储器的初次擦写（系统没有其他方式的情况下）通常通过 JTAG 方式进行。连接系统和相应的 JTAG 编程器，JTAG 编程器可以直接访问 Flash 存储器，模拟 CPU 对于存储控制器进行操作，读/写 Flash 存储器进行擦写操作。关于 JTAG 的相关知识，请参见第 3 章 3.3.1 节。

与 NAND Flash 相比，NOR Flash 的可靠性更高，发生数据错误的概率较低。使用 NOR Flash 存储系统启动程序较为安全。与之相比，使用 NAND Flash 启动系统，需要采用多种技术保证系统启动的可靠性，比如复制多份启动代码，或是增加额外的 ECC 校验位。

NOR Flash 的随机访问特性使得 NOR Flash 支持就地执行（XIP，eXecute in Place）功能，程序可以直接在 Flash 中运行，无需复制到 DDR 之类的 RAM 存储器中执行。当程序在 NOR Flash 中执行时（也就是处理器直接从 Flash 取指），程序执行时需要创建和使用的变量、使用的堆栈等可读可写数据，无法直接写入 NOR Flash（因为 NOR Flash 的数据写入必须首先擦除整个扇区，速度非常慢）。通常的解决办法是在系统编译链接时，由链接器将变量、堆栈、系统堆等动态数据都映射到 RAM 的地址空间，而只将代码段的地址映射到 Flash 的地址空间。另外一点需要说明的是，由于 Nor Flash 的读取延时比较高，通常在 70ns 甚至更高，而且不支持猝发传输（对于有 Cache 的系统而言，行填充的效率是很低下的），这对于高性能处理器而言会是一个很大的性能瓶颈。因此，XIP 技术通常只运用于 MCU 类的应用（这类应用往往主频不高，但对系统成本非常敏感），而高性能应用及主流操作系统都会采用将软件系统全部存放在 RAM 中的内存布局。

Flash 存储器的使用寿命由擦写次数决定。当前 Flash 存储器的 Sector 大多可以擦写十万次以上。使用 Flash 存储器作为系统的文件系统，或是用于记录频繁改变的日志数据时，某一扇区可能被频繁地擦写。此时，Flash 存储器的驱动程序、文件系统多采用磨损检测（Wear Leveling）技术或负载均衡（Load Balance）技术，确保某个特定存储器扇区不会被频繁擦写。

NOR Flash 和 SRAM 具有类似的引脚和访问时序。

2. NAND Flash

与 NOR Flash 相比，NAND Flash 并不提供随机读访问接口。NAND Flash 以页/块（page/block）模式工作，每块数据包含数个页数据，访问数据前要先激活数据所在页。如图 5-47 所示，典型的 NAND Flash 页大小为 2KB。从页中读出数据的速度较慢，因此 NAND Flash 通常设计有页缓冲寄存器（page register）。读取 NAND Flash 数据首先要从 Flash 数据阵列中将相应页数据载入页缓冲寄存器。将页数据载入页缓冲寄存器所需的时间要远远大于从页缓冲寄存器中连续读出数据所需的时间。操作系统的文件系统通常也以页的形式组织，因此，在 NAND Flash 上建立文件系统相对直接和简单。

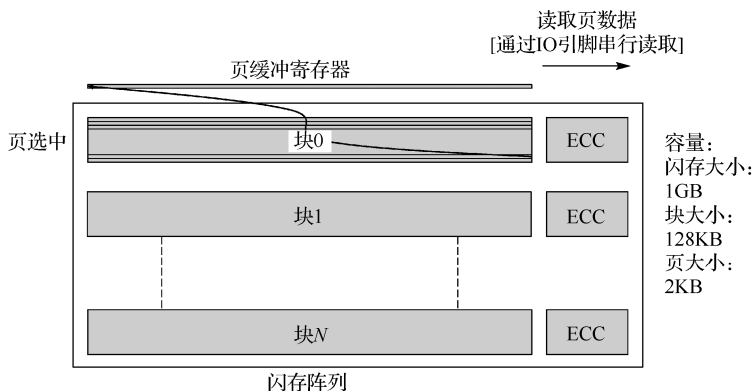


图 5-47 NAND Flash 的组成

常见的 NAND Flash 存储器具有以下特点：

- 以页为单位进行读和编程操作，取决于整个芯片的容量，1 页为 256B、512B 或者更大；以块为单位进行擦除操作，1 块为 4KB、8KB、16KB 或者更大。具有块编程和块擦除的功能，其块擦除时间大约是 2ms（NOR Flash 块擦除时间达到几百毫秒，甚至更长）。
- 数据、地址采用同一总线，实现串行读取。随机读取速度慢且无法进行随机写操作。
- 芯片尺寸小，引脚少，是位成本（bit cost）最低的固态存储器。

NAND Flash 的写操作与 NOR Flash 有所不同，写入数据前先要擦除相应的块（block），对于块的编程必须以串行方式进行，即编程地址必须递增。

NAND Flash 可能会含有坏页。控制器或是存储器驱动必须对坏页进行管理。为了提高 NAND Flash 的产能，存储器可以包含坏页，含有坏页的坏块将被标记，提醒用户避免使用。在 NAND Flash 的使用过程中，也有可能产生坏块。为了保证系统正常运行，NAND Flash 需要添加大量的 ECC 校验位。

NAND Flash 的任意块数据都有可能存在错误，为了实现 NAND Flash 启动，不少 NAND Flash 设计有特殊的启动块。通过增加额外的 ECC 校验位、降低存储密度或是以 NOR Flash 的方式实现，启动块的可靠性可以达到 NOR Flash 的水平。上电后，启动页自动被载入页缓冲寄存器，这样处理器可以直接访问页缓冲寄存器。由于处理器无法直接在 NAND Flash 上执行代码，在处理器启动前，系统将使用 DMA（Direct Memory Access，直接存储器访问）自动将启动页数据载入可随机访问的存储器空间（通常为片上 SRAM，这个搬运过程是上电后硬件自动完成的，因为在 CPU 能够执行代码前，软件什么也做不了），然后启动 CPU，执行可随机访问存储器中的启动代码。启动代码通常在完成主存储器的初始化后（不管是 SDRAM 还是 DDR 都需要初始化，比如关键时序参数的配置等），完成后续代码的搬运工作，即将 Nand Flash 中存放的后续启动代码加载到主存储器（通常是 DDR 或 SDRAM）。接下来运行于主存储器的启动代码将开始加载操作系统。

NAND Flash 出现的初期并没有相应的接口标准。2006 年，开放 NAND Flash 接口（ONFI, Open NAND Flash Interface）行业工作组成立，旨在推动 NAND Flash 的标准化进程。ONFI 工作组建立了一系列的标准规范，包括 ONFI 1.0~3.1 以及 NAND connector 1.0 等，规范详见 <http://www.onfi.org>。

多数 SoC 提供原生的 NAND Flash 接口，这些 SoC 集成有 NAND Flash 控制器，NAND Flash 控制器通过 ONFI 接口对 NAND Flash 进行操作。对于不提供原生 NAND Flash 接口的 SoC 来说，可以通过驱动 GPIO 模仿 ONFI 接口协议对 NAND Flash 进行操作，该操作较原生接口要慢得多。高速 NAND Flash 传输要求 SoC 必须设计有 NAND Flash 控制器。控制器通常支持 DMA，配置后可以自行进行连续数据块传输。

计算机平台上常用的大容量存储器为硬盘。硬盘属于磁性存储器，容量大，价格低。由于硬盘的工作机制、较大的物理尺寸，较高的能耗以及较为严格的工作环境要求限制了其在嵌入式系统，尤其是移动终端中的应用（苹果早期的 Ipod 版本曾经使用过小硬盘，但随着 Nand Flash 的价格下降，几乎所有的产品都只采用 Nand Flash 作为辅助存储器）。

随着 NAND Flash 价格的不断下降，出现了新的存储类型，即固态硬盘（SSD，Solid State Drive）。SSD 由 NAND Flash 以及较为复杂的控制电路组成。控制电路用于管理 NAND Flash 操作，进行磨损检测，同时提供传统的硬盘接口供主机访问。目前主流的硬盘接口规范为 SATA（Serial Advanced Technology Attachment，串行高级技术附件）接口规范，该规范由串行高级技术附件国际组织（Serial ATA International Organization）建立，作为传统并行高级技术附件（Parallel ATA）的演进技术。

图 5-48 所示是一款 NAND Flash 的读操作时序图。在第一个周期给出有效的 CLE 信号和命令 0x00；发送两个周期的列地址；接下来再发送三个周期的行地址；再发一个读操作的第二个周期的命令 0x30；然后 NAND Flash 的内部硬件逻辑根据传入的地址，找到某个块中的某个页，然后把整个这一页的数据都一点点搬运到页缓存中去，此时可以去读取状态位 R/B，是 0 的话就表示系统正在读取数据，是 1 的话就表示已经把整个页的数据搬运到页缓存里去了，最后就可以读取需要的页数据了。

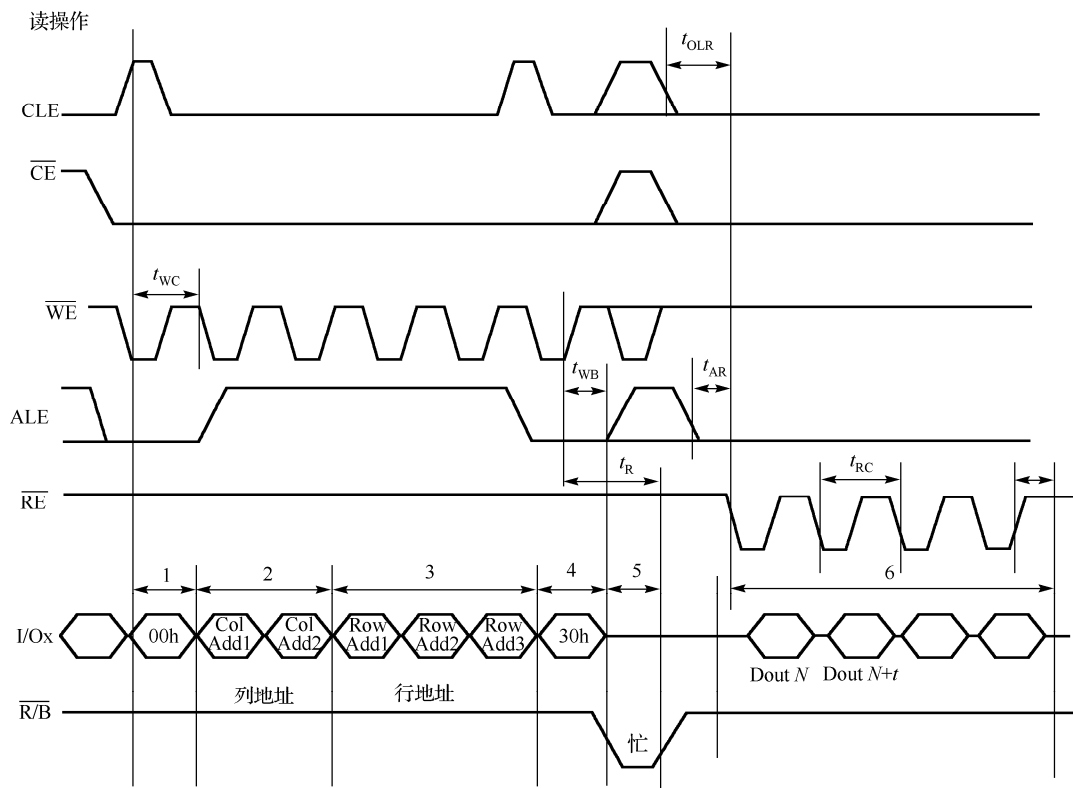


图 5-48 NAND Flash 读操作时序图

图 5-48 中 NAND Flash 各个管脚的意义如下：

- CE: Chip Enable，芯片使能，即 NAND Flash 片选。
- I/O0~I/O7：用于输入地址/数据/命令，输出数据。
- CLE: Command Latch Enable，命令锁存使能。
- ALE: Address Latch Enable，地址锁存使能。

- RE: Read Enable, 读使能。
- WE: Write Enable, 写使能。
- WP: Write Protect, 写保护。
- R/B: Ready/Busy, 主要用于在发送完编程/擦除命令后, 检测这些操作是否完成, 忙表示编程/擦除操作仍在进行中, 就绪表示操作完成。

3. NAND Flash 与 NOR Flash 的比较

NAND Flash 以其较高的容量/价格比在非易失性类存储设备中显现出强劲的市场竞争力, 它的出现为降低产品中存储设备的成本提供了很好的解决方案。NAND 和 NOR Flash 是现在市场上两种主要的非易失闪存芯片, 这两种类型的 Flash 区别在于: NOR Flash 可以按照字节访问, 所以存放在 Flash 里的程序可以直接执行, 而 NAND Flash 是串行访问的, 类似硬盘的存取方式, 需要先把程序读取到内存然后再从内存中运行。

NAND Flash 与 NOR Flash 在性能上的差别主要体现在以下几方面。

1) 性能比较

Flash 闪存是非易失存储器, 可以对存储器单元块进行擦写和再编程。任何 Flash 器件进行写入操作前必须先执行擦除。擦除 NOR 器件时是以 64~128KB 的块进行的, 执行一个写入/擦除操作的时间为 1~5s; 擦除 NAND 器件是以 8~32KB 的块进行的, 执行相同的操作最多只需要 4ms。执行擦除时, 块尺寸的不同进一步拉大了 NOR 和 NAND 之间的性能差距。统计表明, 对于给定的一套写入操作 (尤其是更新小文件时), NOR Flash 需要更多的擦除操作。因此, 当选择存储解决方案时, 设计师必须权衡以下的各项因素。

- NOR 的读取速度比 NAND 稍快一些。
- NAND 的写入速度比 NOR 快很多。
- NAND 的擦除速度远比 NOR 快。
- 大多数写入操作需要先进行擦除操作。
- NAND 的擦除单元更小, 相应的擦除电路更少。

2) 接口差别

NOR Flash 带有 SRAM 接口, 有足够的地址引脚来寻址, 可以很容易地存取其内容的每一字节。NAND 器件使用复杂的 I/O 口来串行地存取数据, 各个产品或厂商的方法可能各不相同。8 个引脚用来传送控制、地址和数据信息。NAND 的读和写操作采用固定大小的块, 这一点有点像硬盘管理此类操作。很自然地, 基于 NAND 的存储器就可以取代硬盘或其他块设备。

3) 容量和成本

NAND Flash 的单元尺寸几乎是 NOR 器件的一半。由于生产过程更为简单, NAND 结构可以在给定的模具尺寸内提供更高的容量, 从而相应地降低了价格。

4) 可靠性和耐用性

采用 Flash 介质时, 一个需要重点考虑的问题是可靠性。对于需要扩展 MTBF 的系统来说, Flash 是非常合适的存储方案。可以从寿命 (耐用性)、位交换和坏块处理三个方面来比较 NOR 和 NAND 的可靠性。

(1) 寿命 (耐用性)。

在 NAND 闪存中, 每个块的最大擦写次数是 100 万次, 而 NOR 的擦写次数是 10 万次。NAND 存储器除了具有 10:1 的块擦除周期优势外, 典型的 NAND 块尺寸是 NOR 器件的 1/8, 每个 NAND 存储器块在给定时间内的删除次数要少一些。

(2) 位交换。

所有 Flash 器件都受位交换现象的困扰。一位的变化可能不很明显，但是如果发生在一个关键文件上，这个小小的故障就可能导致系统停机。如果只是报告有问题，多读几次就可能解决。位反转的问题更多见于 NAND 闪存，NAND 的供应商建议使用 NAND 闪存的时候，同时使用 EDC/ECC 算法。当然，如果用本地存储设备来存储操作系统、配置文件或其他敏感信息，则必须使用 EDC/ECC 系统以确保可靠性。

(3) 坏块处理。

NAND 器件中的坏块是随机分布的。NAND 器件需要对介质进行初始化扫描以发现坏块，并将坏块标记为不可用。

5) 易用性

可以非常直接地使用基于 NOR 的闪存，像其他存储器那样连接，并可以在上面直接运行代码。由于需要 I/O 接口，NAND 要复杂得多。各种 NAND 器件的存取方法因厂家而异。在使用 NAND 器件时，必须先写入驱动程序，才能继续执行其他操作。向 NAND 器件写入信息需要相当的技巧，因为用户绝不能向坏块写入，这就意味着在 NAND 器件上自始至终都必须进行虚拟映射。

6) 软件支持

在 NOR 器件上运行代码不需要任何的软件支持。在 NAND 器件上进行同样操作时，通常需要专门的底层软件支持，对于 Linux 系统而言，这层软件被称为存储技术设备 (MTD, Memory Technology Device)。NAND 和 NOR 器件在进行写入和擦除操作时都需要 MTD。使用 NOR 器件时，所需要的 MTD 要相对少一些。许多厂商都提供用于 NOR 器件的更高级的软件，其中包括 M-System 的 TrueFFS 驱动，该驱动被 Wind River System、Microsoft、QNX Software System、Symbian 和 Intel 等厂商所采用。

4*. SLC/MLC/TLC NAND

NAND 闪存芯片包括 SLC (single-level cell) 类型、MLC (multi-level cell) 类型和 TLC (triple-level cell) 类型。这些类型的 NAND 闪存结构上差不多，但是 MLC 相对于 SLC 具有更高的存储密度，而 TLC 又比 MLC 的存储密度高。存储密度越高相对的单位比特的价格就会越低。所谓 MLC 是相对 SLC 而言的，我们都知道 Flash 的存储原理是存储在浮栅中的电荷。对于 SLC 的器件，浮栅电荷只有两种情况，也就是有电荷与无电荷 (0 和 1)。而对于 MLC 器件而言，浮栅中的电荷量可以分为 4 级，分别对应 00、01、10 和 11。因此 MLC 的一个存储单元就可以存储 2 比特信息，在不增加存储单元的情况下，存储容量可以提升一倍。TLC 器件则可以区分浮栅上的电荷的 8 个级别，分别对应 000、001、直到 111，所以一个 TLC 器件可以存储 3 比特信息。虽然 MLC 和 TLC 在存储密度上具有优势，但是 SLC 相对于其他两种类型的 NAND 具有更佳的读写性能及更佳的可靠性和使用寿命 (意味着可以更多次地承受擦除操作，通常要比 MLC 高 10~20 倍)。相对于 SLC 类型的闪存来说，MLC、TLC 类型的 NAND 具有更低的成本、更大的存储空间，所以更适合于需要低成本大容量数据存取的领域。表 5-6 给出了这几种技术的 NAND Flash 的比较。

表 5-6 几种不同 NAND Flash 技术的比较

	SLC	MLC	TLC
每单元比特数	1	2	3
擦写寿命 (次)	100000	3000	1000
读时间 (微秒)	25	50	~75
写时间 (微秒)	200~300	600~900	900~1350
擦写时间 (毫秒)	1.5~2	3	4.5

5*. SD/MMC 存储卡

SD 卡与 MMC 的存储介质均为 NAND Flash。MMC 规范目前由 JEDEC 组织管理。针对嵌入式应用,还存在 eMMC (Embedded MMC, 嵌入式多媒体卡) 规范。MMC 由存储器与控制电路组成,控制电路将 MMC 接口信号转换为 NAND Flash 接口规范,控制器通过 MMC 接口与 MMC 卡相连,对 MMC 卡内的 NAND Flash 进行读/写操作。SD 卡 (Secure Digital card, 安全数字卡) 在 MMC 的基础上发展而来,MMC/SD 接口规范由 SD 组织 (SD Association) 维护,符合 MMC/SD 规范的控制器的可以正常访问 SD 卡、MMC 以及 eMMC 等设备。

MMC (MultiMedia Card) 卡由西门子公司和首推 CF 卡的 SanDisk 于 1997 年推出。MMC 存储卡有 MMC 和 SPI 两种工作模式。MMC 模式是标准的默认模式,具有 MMC 的全部特性。而 SPI 模式则是 MMC 存储卡可选的第二种模式,这个模式是 MMC 协议的一个子集,主要用于只需要小数量的卡 (通常是 1 个) 和低数据传输率 (和 MMC 协议相比) 的系统,这个模式可以把设计花费减到最小,但性能不如 MMC。MMC 卡目前已逐步被淘汰,取而代之的是 SD 卡。

SD 卡广泛地应用于便携式装置上,如数码相机、电子阅读器、个人数码助理 (PDA) 和多媒体播放器等。拥有高记忆容量、快速数据传输率、极大的移动灵活性以及很好的安全性等特点。SD 卡支持 SD 模式和 SPI 模式两种通信方式。采用 SPI 模式时,占用较少的 I/O 资源。SD 存储卡的物理参数、引脚定义以及数据传输的协议兼容 MMC 卡。SD 存储卡的通信是基于一个高级的九针接口 (时钟线、命令线、四针数据线、三针电源线),这个接口可以工作在最大频率 25MHz 和低电压范围。

对于 MMC 卡的访问需要遵循 MultiMediaCard Spen ver4.X/3.X 协议进行,MMC 卡还有 MMCplus 和 MMCmobile,以及 CEATA specifications (ver1.0) 等标准。对于 SD 卡的访问需要遵循 SD Spec ver1.01/1.10 协议进行。对于 SD 卡和 MMC 卡的数据访问有 1bit/4bit/8bit 3 种模式。对于 SD 卡和 MMC 卡的访问需要遵循协议发送各种命令来进行,比如对于 MMC 卡有 MMCA stream write and read,各个命令的具体意义可以参阅 SD 卡和 MMC 卡的协议。

6*. SDXC 存储卡

SDXC 存储卡不但拥有超高的容量,而且其数据传输速度非常快。SDXC 存储卡最大的传输速度预期能够达到 300MB/s。SDXC 存储卡拥有超高容量,不过其数据安全性能如何暂时还不清楚。

SDXC 存储卡技术指标如下:

- SDXC 存储卡目前的最大容量可达 64GB,理论上最大容量能达到 2TB。
- 支持 UHS 104——一种新的超高速 SD 接口规格,这是新 SD 存储卡标准 Ver 3.00 中的最高标准,其在 SD 接口上能够实现每秒 104MB 的总线传输速度,从而可实现 35MB/s 的最大写入速度和 60MB/s 的最大读取速度。
- UHS104 提供传统的 SD 接口——3.3V DS (25MHz) / HS (50MHz),支持 UHS104 的新 SDHC 存储卡和现有的 SDHC 对应设备相兼容。
- SDXC 存储卡只和装有 exFAT 文件系统的 SDXC 对应设备相兼容。它不能用于 SD 或 SDHC 对应设备。
- 采用最可靠的 CPRM 版权保护技术。
- UHS104 是一种新的超高速接口规格,数据总线传输速率为 104MB/s。这是 SD 新存储卡标准 Ver 3.00 中的最高标准。
- SDXC 存储于 2009 年 4 月被 SD 协会定义为下一代 SD 存储卡标准,目的是为满足大容量存储媒体的不断增长的需求,为丰富的存储应用提供更快的数据传输速率。新 SDXC 存储卡标

准和提供 4~32GB 容量的 SDHC 存储卡标准相比，其所实现的容量可超越 32GB，最大可达 2TB（TB：terabyte，万亿字节，1TB=1024GB）。

5.5 外部存储器接口

前面介绍了嵌入式系统中一些常用的存储器。在 SoC 中需要有一个外部存储器接口(EMI, External Memory Interface) 按照存储器访问时序来访问这些存储器，从而使得 CPU 可以通过存储器控制接口访问常用的存储器进行 CPU 执行指令时所必需的一些数据读写操作。SoC 中的外部存储器控制接口一般支持对 SRAM/NOR Flash、SDRAM、NAND Flash 等存储器的访问控制，由于对 SD/MMC 存储卡的访问控制需要按照 SD/MMC 协议来发送命令，所以一般是单独设计一个 SD/MMC 控制器模块来提供对 SD/MMC 存储卡的访问控制。

5.5.1 SEP4020 芯片的外部存储器接口 EMI^①

1. EMI 的功能

EMI 一般提供 SRAM 控制接口用来产生访问片外 SRAM/NOR Flash 所需要的访问时序信号；提供 SDRAM 控制接口用来产生访问 SDRAM 所需要的访问时序信号；提供 NAND Flash 控制接口用来产生访问 NAND Flash 所需要的访问时序信号。这些存储器所需要的访问时序详见 5.4 节。如图 5-49 所示为 SEP4020 芯片通过 EMI 模块和片外存储器的连接（读者可以通过第 2 章的 2.8.4 节了解 SEP4020 芯片）。

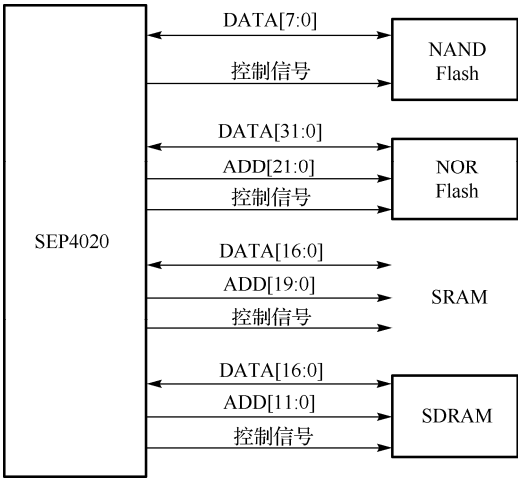


图 5-49 SEP4020 和外部存储器的连接

SEP4020 处理器的 EMI 模块作为一个总线设备连接在片内的 AHB 总线上，统一管理片外存储器，如 NOR/NAND Flash、SRAM、ROM、SDRAM 等。EMI 模块的结构框图如图 5-50 所示，其中 HIU 控制器用于接收 AHB 总线的信号，AFIFO 用于缓存地址信号，DFIFO 用于缓存数据信号，解码器对 AHB 总线发来的地址进行解码，SRAM 接口用于连接片外 SRAM 存储器，SDRAM 接口用于连接片外 SDRAM 存储器，NAND Flash 接口用于连接片外 NAND Flash 存储器。

① 不同的 SoC 芯片可能会采用不同的名字，但该模块的基本功能就是为系统实现对片外存储器，尤其是主存储器的访问和控制。“EMI”是沿用 SEP4020 和 SEP6200 处理器技术文档中的命名。

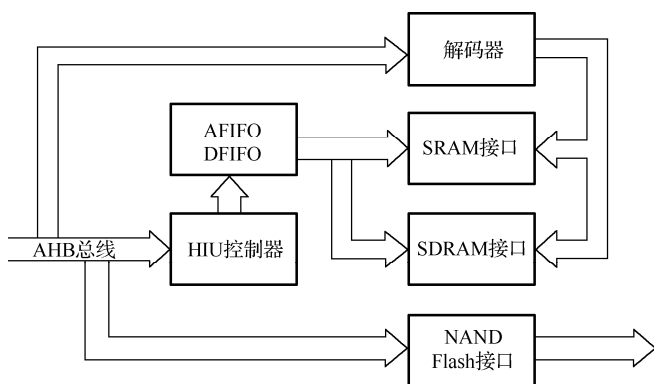


图 5-50 SEP4020 的 EMI 结构框图

EMI 支持的数据传输方式为：SEP4020 EMI 接口模块的 SRAM 接口（CSA、CSB、CSC、CSD、CSE、CSF）支持外部存储器 8 位或 16 位的数据端口，默认情况下为 16 位端口，除了 CSA 外，其他片选可配置为 8 位接口。CSE 和 CSF 两个片选还支持 SDRAM 时序，当配置为 SDRAM 接口时，这两个片选只能用于连接 16 位的 SDRAM 芯片。

图 5-51 所示为如何将 16 位的 NOR Flash 存储器连接到 SEP4020 芯片 EMI 模块的示意图。

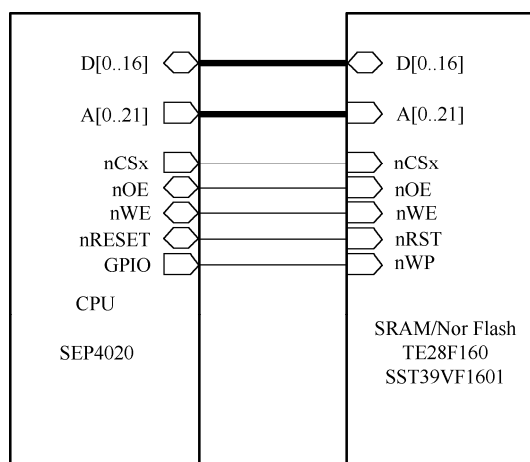


图 5-51 NOR Flash 和 SEP4020 的连接

2. EMI 中的 SRAM/NOR Flash 控制器

SEP4020 中的 EMI 包括 SRAM/NOR Flash 控制器、SDRAM 控制器、NAND Flash 控制器。本小节以 EMI 中相对较简单的 SRAM 控制器为例来讨论接口信号和模块设计。EMI 中 SRAM 控制器模块的设计首先需要考虑接口信号。接口信号包括两类：一类接口信号用于把 EMI 模块连接到总线上，一类接口信号用于把 EMI 模块连接到片外存储器 SRAM。

(1) EMI 中，SRAM/NOR Flash 控制器具有以下特性：

- SEP4020 的片选信号 CSA/CSB/CSC/CSD 用于 SRAM，而 CSE/CSF 根据配置可以用于 SRAM 或 SDRAM。支持 JEDEC 标准的 SRAM 和 NOR Flash 的读操作和写操作。
- 每个片选最大支持 16MB 的地址空间。
- 除 CSA 外，每个片选的数据宽度 8 位和 16 位可配，CSA 仅支持 16 位存储器。

➤ 每个片选的时序参数分别配置，可以使得不同片选所选择的器件工作在不同的时序参数上。这些时序参数包括：OE_HOLD、OE_EN、OE_WAIT、CS_HOLD、CS_WAIT、WE_HOLD、WE_EN、WE_WAIT，如图 5-30 和图 5-31 所示。

(2) SRAM/NOR Flash 控制器接口信号：

SRAM/NOR Flash 控制器的接口信号包括两部分：AHB Slave 接口信号；输出给 SRAM/NOR Flash 的控制信号，如表 5-7 所示。

表 5-7 EMI 的接口信号

AHB Slave 接口信号			
信号名	位数	I/O	描述
hclk	1	I	时钟信号
hresetn	1	I	复位信号
hsel	1	I	操作选择信号
hwrite	1	I	读写操作
htrans	2	I	总线传输类型 idle/busy/noseq/seq
hburst	3	I	总线 burst 长度
hwdata	32	I	写数据
haddr	32	I	地址
hsize	2	I	传输位宽
hrdata	32	O	读数据
hready	1	O	总线响应
hresp	2	O	总线响应
输出给 SRAM 存储器的控制信号			
信号名	位数	I/O	描述
SRAM/NOR FLASH _CSn	1	O	SRAM/NOR FLASH 片选
SRAM/NOR FLASH _Wen	1	O	SRAM/NOR FLASH 写信号
SRAM/NOR FLASH _Oen	1	O	SRAM/NOR FLASH 读信号
SRAM/NOR FLASH _ADDR	n	O	SRAM/NOR FLASH 地址
SRAM/NOR FLASH _DATA	16	IO	SRAM/NOR FLASH 数据
SRAM/NOR FLASH _Ben	2	O	SRAM/NOR FLASH 字节选择

(3) SRAM/NOR Flash 控制器结构：

EMI 中 SRAM/NOR Flash 控制器模块的结构框图如图 5-52 所示，SRAM 和 NOR Flash 的访问时序一致，所以图中以 SRAM 控制器为例。

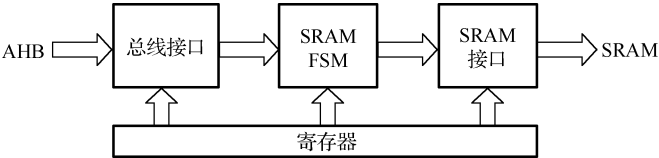


图 5-52 SRAM 控制器结构

- AHB 表示 EMI 连接到 AHB 上的接口信号。
- SRAM 表示 EMI 连接到片外 SRAM 存储器的接口信号。
- 总线接口模块用于处理 AHB 接口信号以及区分寄存器操作、存储器操作。
- 寄存器模块用于控制存储器地址范围、位宽以及控制存储器访问的方式，系统程序员通过配置

这些寄存器来产生管理 SRAM 接口产生的时序。

- SRAM FSM 模块用于处理有效的存储器操作；考虑各种传输类型：数据位宽、读/写；控制输出信号的时序。
- SRAM 接口模块用于根据 FSM 的控制输出相应的信号给 SRAM；匹配总线位宽和 SRAM 位宽。

3. EMI 中的 SDRAM 控制器

SEP4020 的 SDRAM 接口控制器的主要功能如下：

- ① CSE。CSF 可以根据配置用于 SRAM 或 SDRAM。
- ② 支持 JEDEC 标准的 SDRAM。
- ③ 每个 SDRAM 片选最大支持 64MB 的地址空间。
- ④ 每个 SDRAM 片选支持的数据宽度为 16 位。
- ⑤ SDRAM 行地址宽度和列地址宽度可配。行地址范围为 11~13 位，列地址范围为 8~11 位。
- ⑥ 支持 SDRAM 的 Bank 交织（Interleave）访问。
- ⑦ 时序参数可配。时序参数和各自的意义如下：
 - t_{RCD} ：从一行 active 到 read/write 命令等待的周期数。
 - t_{CAS} ：从发出 read 命令到收到数据等待的周期数。
 - t_{RP} ：precharge 命令到下一次 active/refresh 命令需要等待的周期数。
 - t_{RFC} ：从 auto refresh 命令到后续命令需要等待的周期数。
 - t_{RC} ：从 active 一行到 active 另一行需要等待的周期数。
 - t_{XSR} ：从 self refresh 退出时，从 CKE 的上升沿到后续 auto refresh 命令需要等待的周期数。
- ⑧ 一般情况下，其他可以直接满足不需要配置的参数如下：
 - t_{RRD} ：从本次命令的时钟上升沿到发出下个命令的时钟上升沿之间的时间。
 - t_{RAC} ：active 到 precharge 之间的周期数。
 - 等待周期数指的是从发出本次命令的时钟上升沿到发出下个命令的时钟上升沿之间的时间。
- ⑨ SDRAM 的刷新时间可配，每次刷新的行数可配。
- ⑩ SDRAM 的 precharge 不适用 auto precharge 模式，使用 delay precharge 模式。
- ⑪ 支持 SDRAM 的 powerdown mode，一段时间不工作，SDRAM 控制器可以控制 SDRAM 进入 powerdown 模式，进入 powerdown 模式的时间可配。在 powerdown 的情况下，SDRAM 控制器仍然保持对 SDRAM 的 auto refresh。如果 power down 器件有读写访问，则 SDRAM 控制器控制 SDRAM 退出 power down 模式。
- ⑫ 支持 SDRAM 的 self refresh 模式，由 PMU 发出 self refresh 请求。如果有读写请求，自动退出 self refresh 状态。

5.5.2 SEP4020 芯片 EMI 的初始化与配置

对于嵌入式系统的开发者，所关心的不是 SoC 中的 EMI 模块应该如何设计，而是 EMI 模块应该如何使用，也就是如何编写 EMI 的驱动程序（准确地说应该是 EMI 的初始化与配置）从而控制 EMI 的工作。

首先我们要了解 SoC 的地址映射表，从而了解对 EMI 的某个寄存器进行编程是对哪个地址进行读写；另外，我们必须知道 EMI 外接的存储器所占用的系统地址空间是什么。以 ARM 内核为 CPU 的嵌入式系统的启动是从 0x0 地址（ARM720T 以后的 CPU 还可以从 0xFFFF0000 地址开始启动）开始取指令运行，那么 0x0 地址所连接的是哪个片选信号？即哪个存储器？再次，需要了解 EMI 的各个

寄存器的详细功能。本小节以一个寄存器（片选配置寄存器）为例说明如何配置该寄存器以产生 SRAM 存储器所需要的时序信号。

1. EMI 模块中的寄存器

对于嵌入式系统的开发者，控制芯片的操作是通过配置相应的寄存器来完成的，所以在编写 EMI 模块的驱动时，需要先了解 EMI 模块有哪些寄存器，这些寄存器的每一位分别控制什么操作。由于这些寄存器是与存储器地址统一编址的，所以 ARM CPU 可以通过 Load 和 Store 指令对这些寄存器进行读和写。在 SEP4020 芯片中对于 EMI 模块中的 SRAM/NOR Flash 控制器和 SDRAM 控制器设计有下列寄存器，每个寄存器的每一位分别控制 EMI 模块的什么操作请查阅《SEP4020 嵌入式微处理器用户手册》（Ver1.9.0）。其中主要的寄存器包括：

- CSA/CSB/CSC/CSD/CSE/CSF 片选配置寄存器。
- SDRAM 时序配置寄存器 1。
- SDRAM 时序配置寄存器 2。

2. 片选配置寄存器

访问各种片外存储器都需要 EMI 输出给存储器芯片以有效的片选信号 CS。以 SRAM 片外存储器为例，SRAM 接收到有效的片选信号 CS 后，还要根据送给 SRAM 的 OE、WE 和地址信号，进行相应的读写操作。SRAM 需要 CS、OE、WE 和地址等信号以一定的时序送给 SRAM，所以 EMI 就需要提供相应的寄存器来配置 CS、OE、WE 等信号输出的时序。另外，每个 CS 信号对应一段内存空间，EMI 还必须知道该 CS 信号所对应的基地址是什么，这也需要程序员对片选寄存器进行配置（当然，大多数芯片在系统复位的时候都为每个片选设置了缺省的片选基址）。

例如，表 5-8 是 SEP4020 芯片用户手册中 CSA 片选配置寄存器（配置 CSA 片选信号所连接的片外存储器的 CS、OE、WE 等信号产生的时序）各位的意义。

表 5-8 CSA 片选配置寄存器（CSACONF，寄存器基址 0x11000000，偏移量 0x0000）

数据位	名称	类型	复位值 (二进制)	描述
31:30	RESEARVED	—	—	注意： 对于 CSE 和 CSF 两个片选配置寄存器，第 31 位用于标识该片选是 SRAM 时序还是 SDRAM 时序，复位缺省值 1B 表示该片选连接 SDRAM 芯片，用户可将其设置为 0B，以让该片选连接 SRAM 时序的芯片
29:24	CSABA	RW	001000B	CSA 地址区域的基地址高 31~26 位。高地址匹配则产生片选
23:22	WE_HOLD	RW	01B	输入使能从低电平到高电平后保持高电平所需要的时钟周期数 00B: RESERVED 01B: 0 个周期； 10B: 2 个周期 11B: 3 个周期
21:18	WE_EN	RW	1001B	输入使能保持低电平（有效）需要的时钟周期数 0000B: 1 个周期； 0001B: 2 个周期 0010B: 3 个周期； 0011B: 4 个周期 0100B: 5 个周期； 0101B: 7 个周期 0110B: 9 个周期； 0111B: 11 个周期 1000B: 13 个周期； 1001B: 15 个周期 1010B: 17 个周期； 1011B: 19 个周期 1100B: 21 个周期； 1101B: 23 个周期 1110B: 27 个周期； 1111B: 31 个周期

续表

数据位	名称	类型	复位值 (二进制)	描述
17:16	WE_WAIT	RW	01B	片选有效后输入使能保持高电平需要的时钟周期数 00B: RESERVED; 01B: 0 个周期 10B: 2 个周期; 11B: 3 个周期
15:14	OE_HOLD	RW	01B	输出使能从低电平到高电平后保持高电平所需要的时钟周期数 00B: RESERVED; 01B: 0 个周期 10B: 2 个周期; 11B: 3 个周期
13:10	OE_EN	RW	1001B	输出使能保持低电平（有效）需要的时钟周期数 0000B: 1 个周期; 0001B: 2 个周期 0010B: 3 个周期; 0011B: 4 个周期 0100B: 5 个周期; 0101B: 7 个周期 0110B: 9 个周期; 0111B: 11 个周期 1000B: 13 个周期; 1001B: 15 个周期 1010B: 17 个周期; 1011B: 19 个周期 1100B: 21 个周期; 1101B: 23 个周期 1110B: 27 个周期; 1111B: 31 个周期
9:8	OE_WAIT	RW	01B	片选有效后输出使能保持高电平需要的时钟周期数 00B: RESERVED; 01B: 0 个周期 10B: 2 个周期; 11B: 3 个周期
7:6	CS_HOLD	RW	01B	片选从低电平为高电平之后保持高电平的时钟周期数 00B: RESERVED; 01B: 0 个周期 10B: 2 个周期; 11B: 3 个周期
5:4	CS_WAIT	RW	01B	地址有效后片选保持为高电平（无效）的时钟周期数 00B: RESERVED; 01B: 0 个周期 10B: 2 个周期; 11B: 3 个周期
3:1	RESERVED	—	—	注意： 对于 CSA 以外的其他片选配置寄存器，第 3 位用于标识外部存储器的位宽是 8 位还是 16 位
0	EN_CSA	RW	1B	CSA 片选功能使能位。只有使能后，才能使地址译码产生相应的片选信号。 1B: 片选使能; 0B: 片选不使能

SRAM 片选配置寄存器所配置的参数如下。

- 片选功能使能 EN_CSA：当该片选连接了一个有效的存储器时，则将该位配置为 1。
- CSA 片选信号的地址范围 CSABA：配置该片选信号连接的存储器所占用的系统地址空间范围，例如配置地址的 31:26 位为 101010B，则 CPU 发出的地址在 0xA800 0000H~0xABFF FFFFH 范围内，EMI 就会使片选信号 CSA 有效（这里是由高电平变成低电平），并按照所配置的时序参数产生 CS、WE、OE 信号。比如，对于缺省值 OE_WAIT 是 1001B，对应于 15 个周期，这就意味着对于一个 50MHz 主频的系统而言，每时钟周期为 20ns，每次发起读请求的时候，CPU 需要等待至少 300ns（15×20ns）的时间，显然这是一个比较慢的设备。而对于一个读延时为 60ns 的存储器，同样的系统可以设置这个值为 0010B（60ns）或者 0011B（80ns）。
- 连接到存储器的片选信号 CS、读信号 OE、写信号 WE 的时序，如图 5-30 和图 5-31 所示。

在 EMI 驱动程序中通过正确地配置片选配置寄存器，EMI 中的 SRAM/NOR Flash 控制器就可以在 CS、WE、OE 等引脚上按照 SRAM/NOR Flash 存储器的时序要求生成满足要求的 CS、WE、OE 等时序信号送给 SRAM/NOR Flash 存储器。SEP4020 芯片的 6 个片选信号（CSA~CSF）不仅可以用于连接符合 SRAM 读写时序的存储器芯片（比如 SRAM、ROM 和 NOR Flash），还可以用于连接具备该读写接口的外设芯片，比如串口扩展芯片、FPGA、USB Host 控制器等。需要注意的是，当外接芯片

比较多时，由于这些芯片将共享地址总线、数据总线和控制信号（OE，WE），造成总线容性负载的增加^①，有可能造成 SEP4020 的总线驱动能力不足，此时可能需要在数据总线和地址总线上增加总线缓冲器（Bus Buffer）。

SEP4020 芯片的 CSA、CSB、CSC、CSD、CSE 和 CSF 片选信号不能直接用于与 NAND Flash 存储器芯片的连接，如果需要连接到一个 NAND Flash 存储器，需要使用专门的 NAND Flash 控制信号，如图 5-53 所示，同时也需要在 EMI 驱动程序中相应地配置 NAND Flash 的配置寄存器，以在 R/B、nFRE、nFCE、FCLE、FALE、nFWE、nFWP 等引脚上生成满足 NAND Flash 时序要求（如图 5-48 所示）的 R/B、nFRE、nFCE、FCLE、FALE、nFWE、nFWP 等信号送给 NAND Flash 存储器。

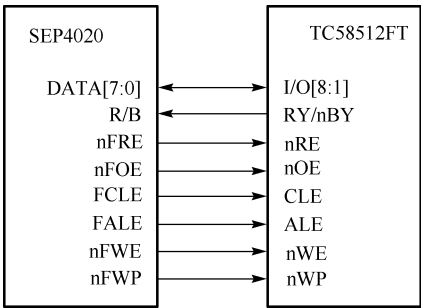


图 5-53 SEP4020 和 NAND FLASH 的连接

3. SDRAM 时序配置寄存器

对 SDRAM 进行操作配置，需要配置片选配置寄存器以及 SDRAM 两个时序配置寄存器，分别为 SDCONF1 和 SDCONF2。片选配置寄存器类似表 5-8（第 31 位用于标识该片选是 SRAM 时序还是 SDRAM 时序）。SEP4020 用户手册中 SDRAM 时序配置寄存器 1（SDCONF1）各位的意义见表 5-9，SDRAM 时序配置寄存器 2（SDCONF2）各位的意义见表 5-10。

表 5-9 SDRAM 时序配置寄存器 1（SDCONF1，寄存器地址 0x11000018）

数据位	名称	类型	复位值	描述
31:29	RESERVED	—	—	—
28:26	TXSR	RW	111B	从 self refresh 退出时，从 CKE 的上升沿到后续 auto refresh 命令需要等待的周期数 000B: 1 个周期；001B: 2 个周期 010B: 3 个周期；011B: 4 个周期 100B: 5 个周期；101B: 6 个周期 110B: 7 个周期；111B: 8 个周期
25:24	SROW		00B	决定 SDRAM 的行地址位数 00B: 11 位；01B: 12 位 10B: 13 位；11B: RESERVED
23:22	RESERVED	—	—	—
21:20	SCOL	RW	00B	决定 SDRAM 的列地址位数 00B: 8 位；01B: 9 位 02B: 10 位；03B: 11 位
19	ADDR_MODE	RW	0B	SDRAM Interleaved Address Mode 控制位 0B: 非 Interleaved Address Mode 1B: Interleaved Address Mode 该位定义了地址的组织，即 bank 地址、行地址、和列地址的顺序。在 Interleaved 模式下，顺序为行地址、bank 地址和列地址，所以当地址递增时是读写完当前 bank 的某行时，将跳到下一个 bank 的对应行读写。而在非 Interleaved 模式下，顺序为 bank 地址、行地址和列地址，所以当地址递增时是读写完当前 bank 的某行时，将对当前 bank 的下一行进行读写

① SEP4020 芯片所能正常驱动的总线负载为 20pF，如果存储器系统设计中只包括一片 NOR Flash、一片 SDRAM 和一片 NAND Flash 芯片，一般不需要增加总线 Buffer。

续表

数据位	名称	类型	复位值	描述
18:16	RESERVED	—	—	—
15:14	SREF	RW	01B	这两位决定 SDRAM 刷新率，即在每个刷新时钟沿到来时，对 SDRAM 做刷新操作的行数。对这两位的位置要参考刷新时钟频率和 SDRAM 参数 00B: 不刷新；01B: 1 行 10B: 2 行；11B: 4 行
13:12	CLKST	RW	00B	这两位决定是否使 SDCLK 进入 power down 模式，将 SDRAM 置于低功耗模式 00B: 不挂起 01B: 当没有 bank 被激活时挂起 10B: 在最后一次访问后 64 个时钟后挂起 11B: 在最后一次访问后 128 个时钟后挂起
11:9	RESERVED	—	—	—
8	SCL	RW	0B	这一位决定 CAS 延迟，即读命令和 SDRAM 数据有效之间的间隔 0B: 2 周期；1B: 3 周期
7	RESERVED	—	—	—
6	SRP	RW	1B	决定对某个 bank 的 PRECHARGE 命令和下一个对同一个 bank 的 ACTIVE 命令之间要插入的时钟个数 0B: 2 个时钟；1B: 3 个时钟
5:4	TRCD	RW	11B	决定 ACTIVE 命令和对同一个 bank 的读/写命令之间要插入的时钟个数 00B: 1 周期；01B: 2 周期 10B: 3 周期；11B: 4 周期
3	RESERVED	—	—	—
2:0	TRFC	RW	111B	这 3 位决定 REFERSH 命令与后续的命令之间需插入的时钟个数 000B: 保留； 001B: 1 个周期 010B: 2 个周期；011B: 3 个周期 100B: 4 个周期；101B: 5 个周期 110B: 6 个周期；111B: 7 个周期

表 5-10 SDRAM 时序配置寄存器 2 (SDCONF2, 寄存器地址 0x1100001C)

数据位	名称	类型	复位值	描述
31	INI	RW	1B	初始化 SDRAM 使能，初始化完成后此位自动清零 1B: 初始化 SDRAM 0B: 不初始化
30:16	RESERVED	—	—	—
15:4	ARFIT	RW	000110000110B	每两次 Auto-Refresh 请求之间的间隔。EMI 做刷新操作之间的间隔的系统时钟周期数，每次刷新操作刷新的行数由 SDCONF1 中的 SREF 决定
3:1	RESERVE	—	—	—
0	REF1	RW	0B	表示 SDRAM 在 self_refresh 模式时，是要刷新 1 行，还是刷新所有的行 1B: 刷新 1 行；0B 刷新所有行

正确连接好片外 SDRAM 存储器各个引脚后，在 EMI 驱动程序中通过正确地配置 SDRAM 控制器的相应寄存器，EMI 中的 SDRAM 控制器就可以在连接到 SDRAM 存储器的各引脚上按照表 5-5 中 SDRAM 指令对应的控制信号真值表生成各控制信号，并且生成的指令能够满足各个 SDRAM 厂商不同 SDRAM 存储器时序的要求，然后即可对 SDRAM 存储器进行正确的访问。

SDRAM 只能接在片选 E 和片选 F 上（这两个片选支持 16 位的 SDRAM，SEP4020 不支持 32 位的 SDRAM），因此配置信息只要在 CSECONF、CSFCONF 两个寄存器以及 SDRAM 两个时序配置寄存器中做相关配置。

5.5.3 OMAP4460 处理器的外部存储器接口

OMAP4460 高性能多媒体应用处理器是由美国 TI 公司推出的基于其 OMAP 架构的一款面向 2.5G/3G 移动智能终端、高性能 PDA 应用的 SoC。该芯片采用 45ns 工艺，其主要功能包括：

- 高清视频播放（1920 × 1080p, 30fps）。
- 2D/3D 游戏。
- 视频会议。
- 高分辨率静态图片（高达 1600 万像素）。
- 支持 Linux、Palm OS、Symbian OS、Windows CE 等高级操作系统。

其主要硬件模块包括：

- 双核 Cortex A9 CPU。
- 数字信号处理器（DSP）子系统。
- 高清图像与视频加速器（IVA-HD）子系统。
- Cortex M3 子系统，包括两个 Cortex M3 微控制器。
- 显示子系统。
- 音频后端（ABE）子系统。
- 图像子系统，包括图像信号处理器（ISP）和静态图像协处理器（SIMCOP）。
- 2D/3D 图形加速器（SGX）子系统。
- 仿真（EMU）子系统。

图 5-54 所示是 OMAP4460 处理器的互联结构和存储子系统的框图（注意，部分细节并未体现在图中），其中 L3 互联和 L4 互联是芯片内部的高速互联结构，L3 主要用于高速设备的连接，而 L4 则主要用于外设的互联，在 L3 和 L4 外设互联直接有 4 个 32 位的数据通路进行连接，L4 配置互联用于对外设的配置，因此只有一个 32 位的数据通路与 L3 连接。高速主设备，比如 DSP，高清视频引擎，图形子系统和图形加速器都有 64 位升至 128 位的数据通路与 L3 连接，而 Cortex A9 双核不仅与 L3 连接，还直接通过 128 位数据线与 DDR 控制器（图中的 EMIF）相连。这样的设计可能是为了尽可能地降低 CPU 的访存延时。

OMAP4460 的外部存储器接口主要由 4 个部分构成，GPMC（General Purpose Memory Controller，通用存储控制器）主要负责 NOR/NAND Flash 的访问，DDR 存储器部分则被分为 DMM（Dynamic Memory Manager，动态内存管理器），主要负责将来自 L3 互联的访存请求进行调度，而 EMIF1 和 EMIF2 则是具体的 DDR 控制器。在 OMAP4460 中可以使用两个物理 DDR 通道，使得系统能够最大程度地降低延时，提高带宽。

该芯片的内存空间被划分为 4 个层次，其中 L1 将 4GB 存储空间划分为 Q0、Q1、Q2 和 Q3 四个区，每个区对应 1GB 空间。L2 层将每个 Q 区进一步划分为 32 个 32MB 的空间。这样就可以通过地址的高 7 位标识共 128 个 32MB 内存空间块（Memory Block）。GPMC 模块共有 8 个独立的片选（gpmc_nc0 ~ gpmc_nc7），这些片选可以用于选择 NOR Flash/NAND Flash 以及 SRAM 芯片。每个片选所对应的地址空间可以配置为 16MB、32MB、64MB 或 128MB。8 个片选共享 Q0 区的 1GB 空间。

Q2 区的 1GB 空间被两个低功耗 DDR（LPDDR）控制器（EMIF1 和 EMIF2）所共享，每个控制器都有自己的片选信号 lpddr21_ncs0 和 lpddr22_ncs0，每个片选都可以配置为 64MB、128MB、256MB、

512MB、1024MB 乃至 2048MB 的空间。两个片选可以配置为交织共享 Q2 区空间，交织的粒度可配置从 128 字节到 512 字节。比如，如果我们配置交织粒度为 512 字节，则地址空间的第一个 512 字节对应于 EMIF1 的片选，而第二个 512 字节则对应于 EMIF2 的片选，以此类推。EMIF1_CS0 在系统复位时的基址是 0x80000000，占据 1GB 地址空间（系统复位时没有地址交织）。

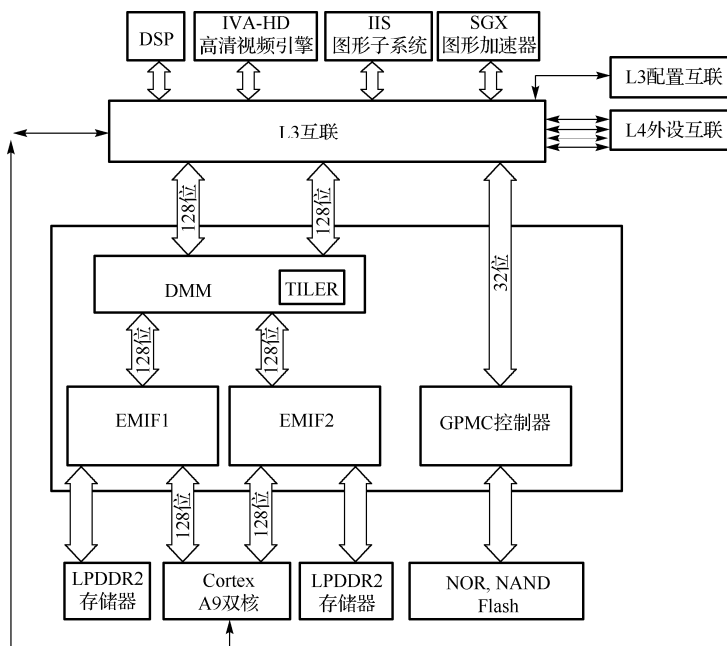


图 5-54 OMAP4460 处理器的外部存储器系统

Q3 区的地址也被映射为 EMIF1 和 EMIF2 的空间，对应的片选信号是 `lpddr21_ncs1` 和 `lpddr22_ncs1`，片选可配置对应的地址空间是 64MB、128MB、256MB、512MB 或 1024MB 空间，两个片选的地址交织粒度是 128MB。系统复位时这两个片选是禁止的，其基址可配，以保证和对应 CS0 片选的地址空间连续。如果 EMIF1-CS1 被禁止，且系统地址不交织，则 EMIF1_CS0 的空间可以被配置为 2GB；或者 EMIF1-CS0 与 EMIF2-CS0 都被配置为 1GB，且两个片选地址交织。

Q3 区的地址空间还被用于分片系统（Tiler System），该空间只被图形子系统（IIS）和显示子系统可见。Q1 区所对应的 1GB 地址空间被分配给系统中所有的外设，包括片内存储器、L3 上所有设备的配置寄存器、L4 上的所有外设寄存器等。

5.6* 存储子系统优化技术

5.6.1 存储子系统的技术指标

随着现代嵌入式系统中 SoC 芯片处理能力的提高，片外存储器（一般为容量较大、能耗较高的 DRAM 芯片）与处理器之间的速度差也越来越大，这一矛盾也被称为存储墙（Memory Wall），如图 5-1 所示。存储墙的存在日益成为嵌入式系统性能、功耗和成本的瓶颈。本节将重点介绍存储子系统的 3 个核心技术指标，即延迟、带宽与功耗。这 3 个技术指标既相互关联又相互冲突，在一款高性能 SoC 的具体设计与应用中，不同处理模块或计算引擎对于它们的要求又各不相同，因此在本节的最后我们还将介绍不同总线 Master 对于存储子系统延迟和带宽的不同需求。

1. 延迟（Latency）

处理器访存的延迟，指处理器通过系统总线发出内存访问请求到该请求被存储系统处理结束（也就是读操作获得数据，写操作完成数据写入）所耗费的时间。处理器访存的延迟和系统总线的传输效率、存储控制系统内部仲裁机制、存储器类型以及存储器控制电路的控制策略有关。由于处理器（或者是其他总线 Master，比如 DMA 控制器、LCD 控制器、GPU、VPU 等模块）通过系统总线发出访存操作时，存储控制系统中内存数据通路可能被其他设备占用，这样处理器的访存时间就不会固定；另外，在使用开页（OpenPage）技术的 SDRAM 内存控制策略中，行（页）是否命中也会影响到处理器访存的时间长度，所以，我们这里所指的延迟是指在一定的条件下，处理器或其他总线 Master 的平均访存延迟。

我们以 CPU 访存为例进行分析。访存延迟包括以下几个部分：CPU 内部排队开销 T_1 （包括时钟域转换的开销）、CPU 到存储器控制器的总线传输开销 T_2 、系统总线请求到 DRAM 命令序列的转换开销 T_3 、存储控制器调度开销 T_4 、DRAM 延迟（PRE+RAS+CAS 延迟） T_5 、存储控制器到 CPU 的总线传输开销 T_6 。因此，总的访存延迟 $T = T_1 + T_2 + T_3 + T_4 + T_5 + T_6$ ，如图 5-55 所示。其中，CPU 内部排队开销 T_1 和总线上传输延迟 T_2 及 T_6 是由处理器体系结构决定的，而协议处理 T_3 、存储控制器调度 T_4 和 DRAM 延迟 T_5 这三部分开销则是由存储控制器的结构决定的。

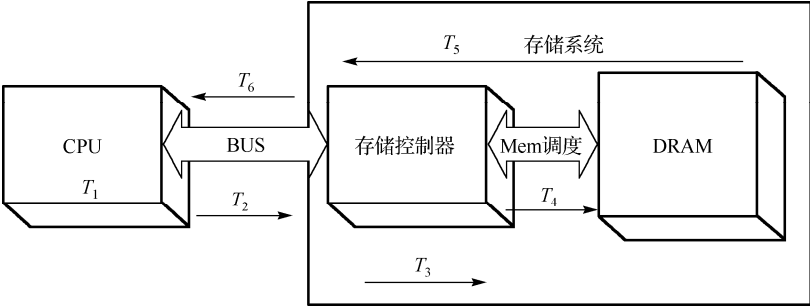


图 5-55 访存延迟的组成

2. 带宽（Bandwidth）

带宽是指 SoC 和存储系统之间有效的数据传送的速率，即单位时间内通过系统总线传输的有效数据的数量，也称为吞吐率（Throughput）。SoC 或者存储控制系统的总线带宽值不但和系统总线的信号交互协议以及系统总线的工作频率有关，而且和存储系统对内存设备的控制策略有密切关系。因此，在系统总线工作频率不变的情况下，为了增加处理器和访存控制系统之间数据传输的带宽，必须提高系统总线的利用率以及提高存储控制系统的访存性能。带宽是存储系统的重要性能指标，它反映了存储系统的整体性能。带宽也是存储系统性能的主要瓶颈，媒体处理程序就是受访存带宽影响最大的一类程序。

下面以一个常见的多个总线主设备访问 DDR 控制器且负载较大的例子，来分析 SoC 芯片中多个主设备对存储系统的带宽需求。该例子是一个常见的基于 Android 系统的媒体播放器，其中，整个系统运行在 Android 操作系统下，VPU 负责解码 1080p 的电影，LCD 负责以 1080p 画质（1920×1080）显示解码后的图像数据，CPU 负责操作系统的运行以及声音数据的解码等。

首先是 CPU 模块，对于 32 位 RISC CPU，由于其本身就有数据缓存和指令缓存从而减少了其对 DDR 带宽的需求。所以，CPU 是一个典型的高优先级、低带宽需求同时又是动态随机访问的主设备。

其次，作为一款多媒体处理器，LCD 模块对 DDR 存储器的访问是必不可少的。如果该 SoC 芯

片的 LCDC 支持 4 层显示，其中的基层显示支持 800×600 的 24 位、18 位或者 16 位的 RGB 图像，常用作背景；覆盖层 1 支持 1920×1080 的 YUV420、YUV422、YUV444 或者 semi-YUV420 的格式，常用于播放视频等；覆盖层 2 支持 800×600 的 24 位、18 位或者 16 位的 RGB 图像；最后一层是鼠标层，支持 1 位或者 2 位的图像。LCDC 与 DDR 接口处的模块框图如图 5-56 所示。

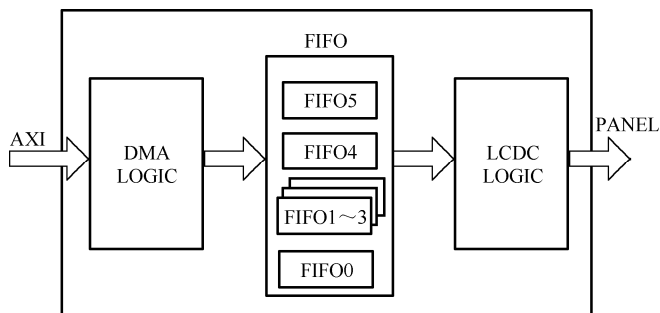


图 5-56 LCDC 内部结构框图

通过框图可以看到，LCDC 与 DDR 控制器的接口使用的是 AXI 接口。这里，由于 LCDC 只会对 DDR 读操作，所以只用到 AXI 接口中的读地址通道和读数据通道，没有使用写地址、写数据以及写响应通道。在支持 4 层显示的 LCDC 模块内部，每一层都有与之对应的 FIFO，其中，FIFO0 对应基层显示，FIFO1~3 对应覆盖层 1 的显示，因为覆盖层 1 支持 YUV 的显示，而在 YUV 格式的图像中，亮度以及色度数据分别存放于不同的地址，所以需要 3 个 FIFO 分别与之对应，FIFO4 对应于覆盖层 2，FIFO5 对应于鼠标层。当 LCDC 向 DDR 控制器请求数据时，各个 FIFO 之间的请求通过不同的读 ID 区分。

LCDC 模块对 DDR 的负载需求较大，最大的情况出现在播放 1080p 电影时（假设 LCD 的分辨率支持 1080p，不需要缩放 Scaling Down），此时，LCDC 对 DDR 总线的数据需求量为： $Q = 1920 \times 1080 \times 1.5 \times 60 = 178\text{MB/s}$ 。

其中，1.5 表示每个像素占用的大小是 1.5 字节（以 YUV420 为例），60 代表 LCDC 每秒至少刷新 60 帧。这还只是采用 YUV 格式的视频数据，如果是 24 位的 RGB 数据，LCDC 的带宽需求将达到近 360MB/s。

再者，分析 VPU 对 DDR 控制器的极限带宽。SoC 芯片中的 VPU 模块不仅支持 1080p 的高清解码，还支持缩放等常见的后处理功能。VPU 模块在系统中的框图如图 5-57 所示。

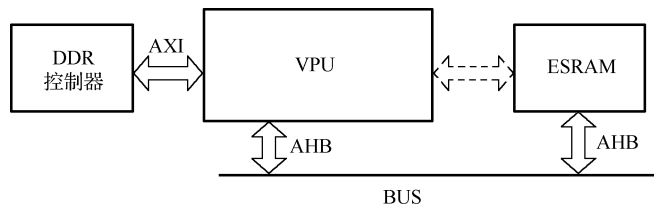


图 5-57 VPU 与总线连接框图

通过图 5-57 可以看出，VPU 模块主要通过 AXI 总线和 DDR 进行数据交换，处理器可通过 AHB 总线对 VPU 配置，同时，VPU 还可以被配置为与整个 SoC 系统共享 ESRAM（片上静态存储器）用作帧缓冲区。在 VPU 解码时，VPU 通过 AXI 总线从 DDR 中取数据，当 VPU 解码完成后，解码的输出数据也是通过 AXI 端口将数据写入 DDR 中。如果需要做后处理，以缩放为例，VPU 会通过 AXI

端口从 DDR 中读取需要缩放帧的 YUV 图像数据，然后经过处理后再将其再写回到 DDR 中。所以，VPU 对 DDR 负载需求最大的情况出现在 VPU 同时解码高清电影且缩放图像时（一般是由于外部的 LCD 显示屏分辨率有限的原因）。以解码 H.264 的高清电影、然后缩放至 VGA 画质为例，此时 VPU 对 DDR 的带宽需求为： $Q = (1920 \times 1080 \times 1.5 \times (2 + 0.01) + 640 \times 480 \times 1.5) \times 30 \approx 195\text{MB/s}$ 。

其中，1.5 表示每个像素占用的大小是 1.5 个字节（以 YUV420 为例），2 代表这幅图像被解码器输出，同时又被后处理模块读取，这两次操作产生的带宽，0.01 代表 H.264 的压缩率，最后的 30 表示 VPU 每秒解码的图像帧数为 30 帧。

以上分析了在一个典型的大数据负载场景下各个主机设备对 EMI 的带宽需求量，从上面的分析可以看出，CPU 虽然对 EMI 的带宽需求不像 VPU 和 LCDC 那么大，但是对于延迟的要求较高，而类似 VPU 和 LCDC 这样的多媒体模块对 DDR 控制器的带宽需求较大。

3. 功耗（Power Consumption）

对于存储子系统的功耗，过去的研究工作将重点放在如何降低 CPU 功耗上。而当前的大量研究表明，嵌入式系统中，CPU 能耗在总能耗中所占比例有限，各类存储器与设备往往是嵌入式系统能量消耗的主要因素。在面向影像、视频等的应用中，能耗的主要来源并非数据通路和控制器，而是对存储器的频繁访问。图 5-58 给出了笔记本电脑中各部分功耗在总功耗中所占比例：系统中功耗最高的是显示系统，但硬盘与存储器的总功耗超过系统总功耗的 20%，仅次于显示系统，超过了 CPU 功耗。表 5-11 列出了东南大学国家专用集成电路系统工程技术研究中心开发的 PMP（Portable Media Player）系统平台上各部件的功耗情况。Active 状态时，系统播放视频文件；Idle 状态时，系统不运行任何程序，且背光关闭。在两种状态下：微硬盘的平均功耗都是最高的；SDRAM 平均功耗所占比例也较高，在 Idle 状态下超过了处理器的平均功耗；存储系统的总功耗已远远超出各其他子系统的功耗。闪存往往被认为是功耗较低的非易失存储器件，然而对其进行写操作的功耗要远远高于对其进行读操作的功耗。读写功耗的差异性使得在写操作所占比例较高的应用场合，闪存功耗并不能被忽视。从以上分析不难得出结论：嵌入式系统中，存储系统在总功耗中所占比例较大，降低存储系统功耗对于降低系统整体功耗至关重要。

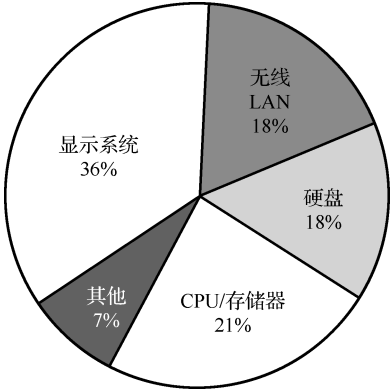


图 5-58 笔记本电脑中各子系统占总功耗的比例

表 5-11 PMP 系统运行时各部件功耗及在总功耗中所占比率

	系统 Active 状态		系统 Idle 状态	
	平均功耗/mW	在系统中所占比例	平均功耗/mW	在系统中所占比例
处理器（Intel XScale PXA255 400MHz）	546	22.6%	121	14.5%
主存（Winbond W982516CH-75L×2 共 64MB）	396	16.4%	193	23.1%
3.5" LCD 触摸屏	77	3.2%	77	9.2%
背光	354	14.7%	0	0.0%
1.8" 微硬盘（HITACHI Travelstar C4K40-20 20G）	711	29.4%	325	38.9%
其他	331	13.7%	119	14.3%
总功耗	2415	—	835	—

4. 不同总线 Master 对存储子系统的不同需求

正如前面的分析,不同的总线 Master 对于存储子系统的延迟与带宽需求是不同的,这种不同的需求来源于这些总线 Master 的不同访存特征。学术界和工业界已经越来越清晰地认识到传统 SDRAM 控制器对于来自总线(或片上网络 NOC)的访存请求不加区分的调度策略(关于这方面调度策略的研究与优化参见 5.6.2 节)已经难以满足高性能处理器对于存储子系统的需求。因此,面向 QoS (Quality of Service, 服务质量)的 DDR 控制器调度策略已经成为研究的热点之一。

所谓 QoS 调度是指针对不同 Master 的访存特性,DDR 控制器在实施访存调度时尽可能满足不同 Master 对存储子系统的不同需求。那么总线 Master 对于存储子系统到底有哪些不同的需求呢?本节将重点分析不同的访存需求,并将这些不同需求归纳为:延迟敏感型需求、带宽敏感型需求和期限敏感型需求。本章的案例将分析高性能低功耗高清媒体 SoC 的 QoS。

延迟敏感型需求。通常一旦 CPU (包括 Cache 发起的)发出访存请求,在等待存储子系统完成这个访存请求前,CPU 只能停下所有操作等待结果。在这种情况下,存储子系统的延迟将直接影响 CPU 的执行性能;而相对 GPU 和 VPU 来说,CPU 对外部存储器带宽的需求要小得多(这一方面也是因为 Cache 的作用)。因此我们把 CPU 的访存特性归结为延迟敏感型需求。我们这里所说的延迟是指系统的“有效”延迟,也就是从 CPU 发出访存请求到完成请求的时间,而不是从 DDR 控制器发出访存请求到完成请求的“裸”延迟。显然“有效”延迟要比“裸”延迟时间长,这是因为多个 Master 对总线的竞争以及总线仲裁需要额外的时间。

带宽敏感型需求。与 CPU 不同,GPU 等总线 Master 往往对存储子系统的带宽有着比较高的需求,而对于延迟并不敏感。这是因为 GPU 通常对大量的数据单元进行相同(或类似)的操作。这些数据单元可能是渲染操作时的像素,也可能是几何处理时的三角形(有时是多边形)。这些数据处理相互独立,也就是说 GPU 不需要等待前一个处理单元的运算结果。如果前一个运算出现访存延迟,GPU 可以直接处理下一个单元而无需等待。VPU (视频处理单元)的访存特性与 GPU 类似。

期限敏感型需求。LCD 控制器(或者 HDMI 控制器)需要从显示缓冲区中不断地将数据扫描输出到显示设备。这个过程必须保证数据输出的恒定速率,任何超过一定期限的延迟都将造成崩溃性的后果(如果 LCD 控制器的输出 FIFO 发生下溢,LCD 屏上显示的画面将出现混乱)。因此对于 LCD 这样的总线 Master 而言,存储子系统的延迟必须满足一定的期限要求,如果延迟超出这个期限,将产生无法接受的系统错误。在国家工程中心自主设计的 SEP4020 处理器中,就曾经出现过类似的问题。如果用户对于 Nand Flash 控制器的时序参数配置不合理,则有可能导致 Nand Flash 的数据传输占用了大量的总线交易时间,使 LCD 控制器在 LCD 屏行频或帧频期限到达后依然无法获取相应的数据,导致 LCD 屏的显示出现左右偏屏或上下偏屏的问题。期限敏感型的总线 Master 与延迟敏感型的总线 Master 都对存储子系统的延迟有要求,但与延迟敏感型需求不同的是,期限敏感型的设备只要求访存延迟必须满足一定期限的要求,而一旦这个期限得到满足后,进一步降低访存延迟对于期限敏感型的设备性能并无明显提高。

5.6.2 DDR 控制器的优化

本小节主要介绍针对存储系统的性能、功耗、带宽问题,DDR 存储控制器的优化方法。

1. DDR 存储控制器的总体架构

访存设备发起访存交易,DRAM 存储器为访存设备提供其所需的数据。访存交易必须通过存储控制器完成访存交易的调度、地址映射、命令翻译等一系列操作。对于访存设备来说,存储控制器可将 DRAM 存储器的访问细节进行屏蔽,从而简化访存设备接口的设计。对于 DRAM 存储器,访存设备

的访存交易请求由存储控制器将其转换成相应的 DRAM 访问命令，并按照时序要求发送给 DRAM 存储器。由于现代 DRAM 存储器的访存效率与访存命令的顺序和访存地址具有非常密切的关系（比如是否需要 Active 操作、是否需要 Precharge 操作等），因此，在不影响程序正确执行的前提下适当调整访存命令的顺序，可以有效地提升访存性能。此外，多个访存设备共享同一主存时，DDR 控制器还可以协调各个设备间的竞争，从而使 DRAM 存储器能够更好地为各个访存设备提供访存需求。

DDR 控制器主要由 3 部分组成：总线接口、调度器和物理层接口，其基本架构如图 5-59 所示。

总线接口是存储控制器与访存设备沟通的桥梁，总线接口首先负责将外部的总线交易进行解析，并将其转换为内部物理层可识别的命令，接着负责将解析后的命令发送到调度器进行相应的调度。此外，总线接口也接受由调度器返回的数据，再按照上层逻辑所需要的时序要求发送给上层用户逻辑。如果这种转换的效率不高，将会直接导致访存性能的下降，无法有效利用访存带宽。在现代 SoC 中，多数的存储控制器均具有多个总线接口，以减少总线冲突对访存性能的影响。

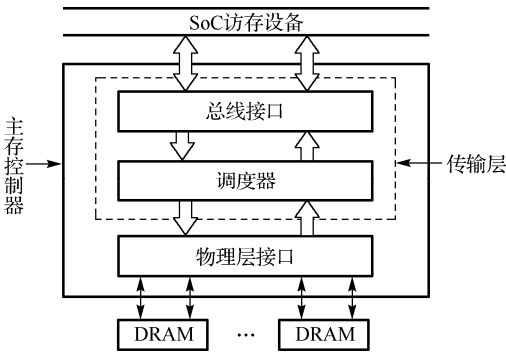


图 5-59 存储控制器基本结构

在总线接口之下是调度器。调度器可以对访存交易的执行顺序进行调整，以优化访存性能。具体而言，其调度优化目标主要有两个：第一个目标是面向访存服务质量（QoS），根据不同访存设备的需求和特性，对各个设备的访存交易先后次序进行调整。通常情况下，这个层次的调度发生在访存交易这个级别，也就是调度器根据从交易等待队列中选择优先被执行的交易。当然，只有总线和访存设备支持 Outstanding 传输时（也就是设备可以在发出访存交易请求后，不必等待该交易完成就发起下一次访存交易请求。相应地，交易的完成顺序也可能与交易的请求顺序不一致。Outstanding 传输其实是在总线上实现的一种乱序机制），交易级的调度才有可能实现；第二个目标是面向 DRAM 的访问效率，通过调整访存命令发出的先后次序，使访存交易尽可能多地为行命中（row hit），并尽可能减少 bank 冲突以及其他转换所导致的额外开销，如对 bank 不同行的访问以及读写命令的转换等，从而提升 DRAM 的带宽利用率。我们有时也将第一种类型的调度称为宏观调度（交易级），而将后一种调度称为微观调度（命令级）。同时，这一部分还负责将物理内存地址转换成相应的 DRAM 通道地址、Rank 地址、Bank 地址、行地址和列地址。总线接口和调度器构成存储控制器的传输层。

在调度器之下是物理层接口。物理层（PHY 层）负责将 DRAM 命令转换成符合 DRAM 接口协议和时序要求的信号组合，发送给外部 DRAM 存储芯片，同时也将接受存储器反馈回来的数据，并将数据返回给传输层。物理层接口的设计与外部 DRAM 存储芯片的类型密切相关，优秀的物理层接口设计能够在保证数据正确传输的同时，充分发掘 DRAM 存储器的性能。

2. DDR 存储控制器地址映射机制

访存命令的地址主要包含 Rank、Bank、Row 和 Col 这些信息^①，传统的地址映射机制是将该地址按照 Rank、Bank、Row 和 Col 从高到低依次排列。当以猝发模式访存时，先访问某一 Col（列），再

① 对于嵌入式系统而言通常很少用到 Rank 的概念，通常高性能 SoC 会提供多个物理访存通道（Channel），也就是多个 DDR 控制器，因此对于这样的系统，其物理地址可以被分为 Channel、Bank、Collum 和 Row。对多个物理访存通道感兴趣的读者可以参与本书的 5.5.3 节。

访问同一 Row（行）的下一列，当某一行的各列都访问完成后，再访问某 Bank 中的下一行，当 Bank 中的所有行访问完成后，再访问某 Rank 中的下一个 Bank。

图 5-60 为经典的地址映射机制。该机制容易使地址连续的访存命令访问同一 Rank 中相同 Bank 的不同行，这就需要预充电关闭当前行并激活下一行，才能访问下一行，因此会导致极大的访存延迟，降低存储系统的性能。



图 5-60 传统地址映射机制

为了降低访存延迟，人们又提出了几种不同的常用地址映射机制：

(1) 页交叉（page interleaving）地址映射机制。

该映射机制是将 Rank 地址置于最高位，Row 和 Bank 的位置与传统地址映射机制相反，即 Row 在高位，Bank 在次高位。在这种地址映射方式下，当以猝发模式访存时，是先访问某一列，再访问同一 Bank 同一行的下一列，当这个 Bank 的各列都访问完后，再访问同一行的下一个 Bank。

这种方式使访存时访问不同 Bank 的概率增加，通过增加 Bank 并行性，可降低访存延迟，如图 5-61 所示。但是，它仅仅是把任意 Rank 中的负载平衡到不同的 Bank 中。



图 5-61 页交叉地址映射机制

(2) Rank 交叉（rank interleaving）地址映射机制。

如图 5-62 所示，该地址映射机制是将 Rank 地址置于 Row 地址和 Bank 地址之间，其他地址排列方式与页交叉模式相同。这种机制由于 Rank 地址较低，因此可以将访存请求分配到不同 Rank 的不同 Bank 上，相对于页交叉模式，Bank 的并行性进一步增强。



图 5-62 Rank 交叉地址映射机制

(3) 基于异或页交叉（permutation-based page interleaving）地址映射机制。

如图 5-63 所示，该地址映射机制是将 Row 地址的部分位与 Bank 地址进行异或操作后得到新的 Bank 地址，从而可以将地址空间相邻的逻辑页映射到多个 Bank 上，降低 Bank 冲突的概率。该策略同时保留了页交叉映射方式的优点。

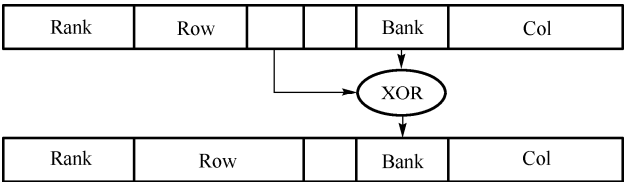


图 5-63 基于异或页交叉模式的地址映射机制

(4) 位翻转（bit-reversal）地址映射机制。

该映射机制是将 Rank 地址位、Bank 地址位以及部分的 Row 地址位进行翻转，其中行地址位翻转位数为翻转深度 v 。研究表明，当其 cache 标签宽度小于 1 时，性能最优。该方式的目的是以提高数

据访存的 Bank 并行性和 Row 局部性为目的，从而减小访存延迟，优化存储系统性能。该地址映射机制如图 5-64 所示。

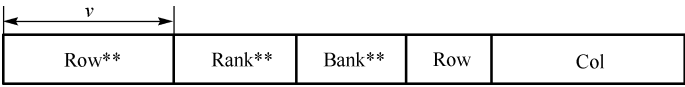


图 5-64 位翻转地址映射机制

3. DDR 控制器访存调度的基本原理及调度策略

1) 访存调度的基本原理

如图 5-65 所示，该图为 DDR 控制器的抽象结构图。当 CPU、GPU 和 I/O 端口等设备发起访存请求时，这些访存请求通过地址映射逻辑将其转换为一系列的访存命令，这些访存命令存放在存储控制器的命令队列中，经过命令调度器的调度后进行分派发送，然后将所访问的 DRAM 地址处存放的数据返回，访存请求得到响应。

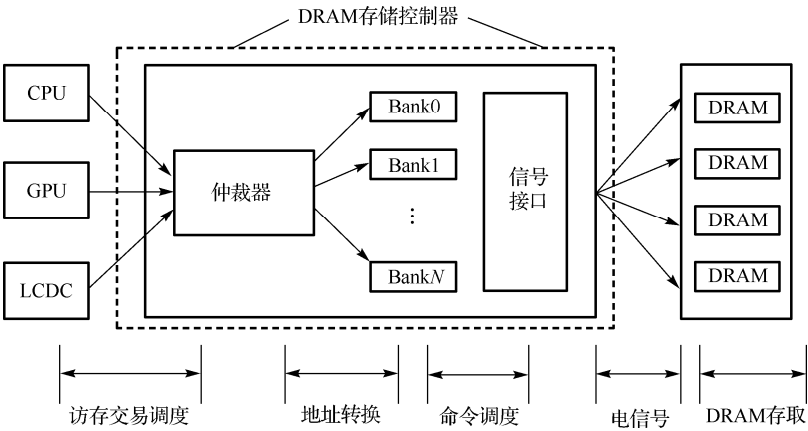


图 5-65 存储控制器抽象结构

仲裁器和命令调度的功能不同。仲裁器是将接收到的外部模块的请求按照特定的仲裁算法进行计算，找出优先级最高的请求。仲裁器一般完成 3 个步骤：访存请求排队、访存请求分类、访存请求仲裁。而调度器是把访存命令按照特定的调度策略重新进行调度，调度的目的是尽量使带宽利用率增大，同时降低访存延迟。

由此可知，DDR 控制器主要完成两部分操作，一部分是对端口访存设备的仲裁，另一部分是对内部访存命令的调度。前者是为了保证服务质量（QoS, Quality of Service），可认为是宏观调度；后者是为了提高 DRAM 的访问效率，可理解为微观调度。

(1) 宏观调度。

正如我们在 5.6.1 节所介绍的，对于多端口设备访存，端口访存类型可分为：延迟敏感型，比如 CPU；带宽敏感型，比如 GPU；期限敏感型，比如 LCD 控制器。当不同类型的主设备并发访问 DRAM 芯片时，对于延迟敏感型设备，应该尽量减少该类型设备的平均访存延时；对于带宽敏感型设备，应该尽量将访存的读写内存 Bank 调度到一起，降低反复换行引起的预充电及行激活开销，充分利用猝发传输特性。此外，通过读取和计算期限敏感型主设备的最高延时容忍，在保证期限满足的前提下，尽量将期限敏感型访存请求调度到存储控制器的空闲时间执行，从而减少对延时敏感型、带宽敏感型访存命令的打断，最终优化 SoC 芯片访存的有效带宽，从而提升存储系统的整体服务质量。

所以，内存访问的宏观调度就是为满足不同类型端口设备的访存需求，对相应的端口访存优先服务的一种调度方式。

(2) 微观调度。

对于一系列内存访问命令，调度器在每个时钟循环内将会选取一个或多个内存命令执行。一种最简单也是最常用的调度算法是：调度器每次选取最长时间未被选择的内存命令。这样内存访问命令就是按序(in-order)执行的，即与内存访问命令到达的顺序是相同的，这种方式是没有经过调度器对命令的顺序进行任何调度的。这可能会导致较大的访存延迟。从图 5-66 中可以看出，同样 8 次内存访问，图 5-66(a)中的访存命令是未经任何调度的，这样，由于相邻的两次内存访问都会发生 Bank 冲突，因此每次访存都需要执行 Bank 预充电(Precharge)、行激活(Row Activation)和列访问(Column Access)命令，因此 8 次内存访问共消耗了 56 个时钟周期。而图 5-66(b)是内存访问命令经过重新排序后的执行情况，调度后，内存访问的 Bank 冲突数明显减少，这时 8 次内存访问只占用了 19 个时钟周期。图 5-66 中假设 Bank 预充电和行激活命令均需要 3 个时钟周期，列访问命令需要 1 个时钟周期。而图中的 (0, 0, 0) 表示本次内存访问的位置为 Bank 0, Row 0, Column 0。

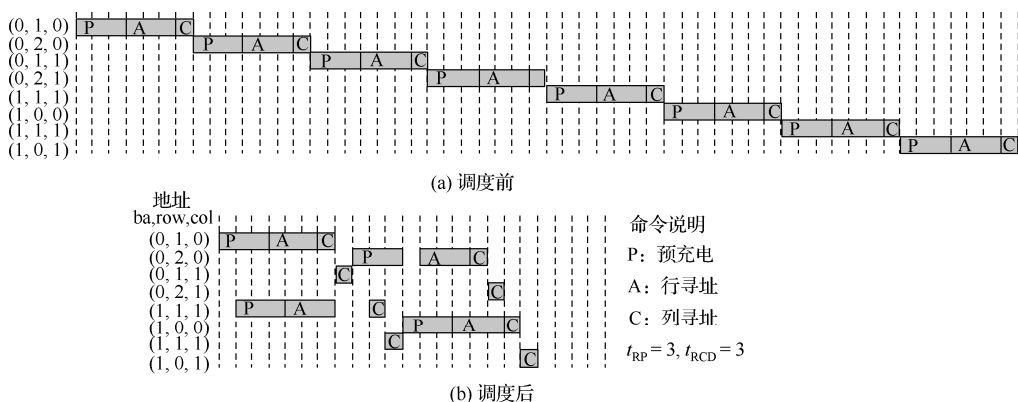


图 5-66 未经调度与经过调度的内存访问延时比较图

因此，内存访问的微观调度是对 DRAM 操作命令的重新排序，这些需要重新排序的命令有：Bank 预充电命令、行激活命令以及列访问命令。图 5-67 给出了存储控制器的 DRAM 命令调度器架构。当内存命令到达后，为该命令分配存储空间并且使该命令处于等待调度器调度的状态，由图 5-67 可以看出，访存命令初始是被置于 DRAM 的 Bank 中的，每个等待调度的命令包含 5 条信息：有效位(V)、加载/存储(L/S, Load/Store)、地址信息(Row 和 Col)、数据(Data)以及状态信息。每个 Bank 都有其单独的预充电管理单元和行仲裁器。Bank 预充电单元决定了何时向相应的 Bank 发射预充电指令，行仲裁器则是当 bank 处于空闲时，决定对该 Bank 中的哪一行进行行激活操作。列仲裁器为所有 Bank 所共享。最后，将由预充电管理单元、行仲裁器以及列仲裁器所选择的命令发射到地址仲裁器中，从而将访问相应的 DRAM 内存。

2) DDR 控制器调度策略

DDR 控制器的调度策略对访存延迟具有很大的影响，因此采用合适的调度策略对优化存储系统性能具有重要的意义。如今，在存储系统中，比较有代表性的主要有 5 种调度策略。

(1) 顺序调度策略。

顺序(In-Order)调度策略是将所有的访存请求按照进入请求队列的时间先后顺序进行排队，所有访存命令按序执行。这种策略没有“饿死”情况的产生，但是由于没有利用 Row 局部性以及 Bank 并行性，因此性能极低。

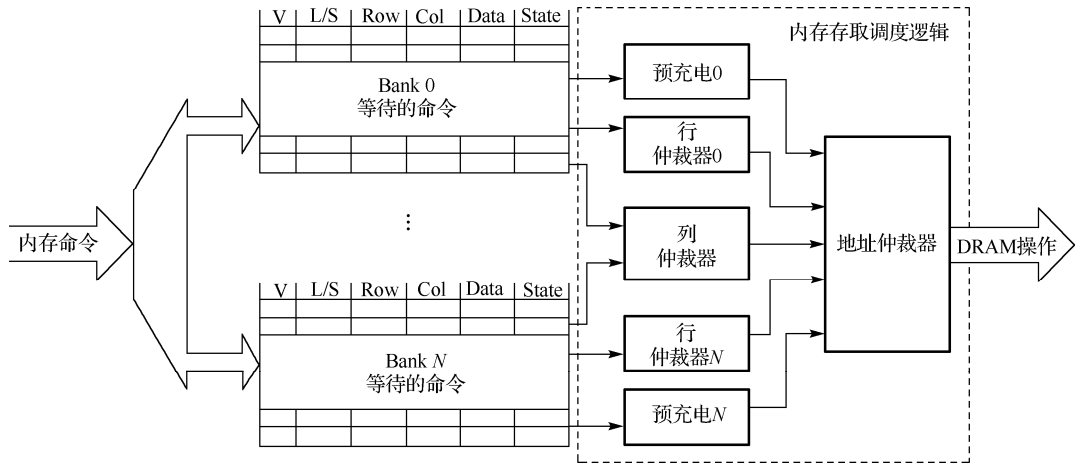


图 5-67 命令调度器架构图

(2) 先到先服务策略。

先到先服务（FCFS，First Come First Serve）是根据访存请求到达主存控制器的时间先后次序来进行服务请求响应的，即先到达的访存请求优先响应。这种策略设计方法简单，一定程度地保证了多线程情况下线程间访存的公平性，且无需额外的硬件开销。但是它没有利用行缓冲命中来减小访存延迟。

(3) 行缓冲命中优先策略。

行缓冲命中优先（FR-FCFS，First Ready First Come First Serve）策略是在 FCFS 的基础上进行的优化策略。访存请求的服务顺序是按照如下要求决定的：①优先处理行命中的请求；②按照请求到达的时间先后顺序响应请求。该策略可以有效减小访存延迟，但是缺少考虑多线程访问的公平性问题。

(4) 最少请求调度策略。

最少请求（Least Request）调度策略是将多个线程中拥有最小访存等待时间的线程置为最高优先级，该策略亦会发生“饿死”现象。

(5) 轮询调转调度策略。

轮询调转（Round-Robin）调度策略在处理多核多线程处理器的存储调度时，可以使各个处理器核的访存请求公平，算法简单。但是对于不同优先级的应用程序，性能的发挥无法得到保障。

5.6.3 片上存储器布局优化技术

片上存储器（Cache/SPM）布局优化技术的目标是将频繁访问的数据放到访问速度快的片上存储器中，从而减少对访问速度慢的片外存储器的访问，以提高系统的性能并且降低功耗。

作为集成在 SoC 内部的 SRAM 存储器，SPM 由于访问速度快，相比 Cache 占用的芯片面积少、能耗低而被广泛应用于当代嵌入式系统中。与对程序员透明的、完全由硬件管理的 Cache 不同，对于程序员来说 SPM 是可以直接进行访问的，因此程序员需要通过软件管理的方式，显式地处理数据在内存和 SPM 之间的关系。因此，SPM 也被称为“软件控制的 Cache”（Software Controlled Cache）。

由于 Cache 和 SPM 这两种片上存储器在管理方式上有着很多不同之处，因此优化算法也有较大的区别。对于单纯采用 SPM 作为片上存储器的嵌入式系统，由于传统编译器认为程序（进程）具有独立的、无访问区别的 4G 连续地址空间，其广泛采用的诸如循环展开、内联核心函数，重新布局通用寄存器等优化技术无法充分发挥 SPM 的优势。基于 SPM 进行布局优化的关键问题在于：对于有限的

SPM 空间，如何确定哪些程序地址空间在什么时候应该被换入 SPM 中，从而最大限度地提高程序的运行速度并降低系统的能耗。

接下来首先讨论单纯采用 SPM 作为片上存储器的嵌入式系统中的 SPM 静态优化技术和 SPM 动态优化技术，然后讨论 SPM 和 Cache 共存架构的布局优化技术。

1. SPM 静态优化技术

SPM 静态优化技术是利用分析程序的代码或数据，找出执行中的热点，并从结果中选取特定的程序子集，将这些程序子集分配到 SPM 中。这些程序片段包括代码或者数据，被称为存储对象 (Memory Object)。在程序运行期间，程序子集一直储存在分配到的 SPM 中，不发生改变。通过静态分配 SPM 空间以容纳部分访问频繁的存储对象是提高系统性能、降低系统能耗十分有效的方式。

通过对程序语言中不同变量的生存时间和存储时间的分析可以得出，程序的指令段及全局变量是最适合被静态分配到 SPM 中的。指令段与全局变量会在整个程序的执行周期内存在，正好满足静态分配所能处理的范围。因此早期的 SPM 分配策略主要针对有着相同的生存周期的程序指令段与全局数据变量。

对于常量和全局变量，一般可以通过静态分析（通常是基于程序控制流图 CFG 和数据流图 DFG）与统计得到的访问次数，然后通过特定的优化算法（一般是将其抽象为背包问题并建模）来决定是否将其分配到 SPM。对于数组，则是分析数组的大小、生命周期以及访问冲突等特性，然后选择出适合分配到 SPM 的数组或者数组的一部分。

2. SPM 动态优化技术

SPM 静态优化技术主要关注于解决部分程序指令、常量数据以及全局数据的优化问题。然而由于被分配到 SPM 的存储对象在程序执行的整个过程中将一直存在 SPM，这些存储空间只能被这些存储对象所独占，其他的存储对象则无法被优化，从而最终导致静态的分配算法对 SPM 的利用率有限。动态 SPM 优化将存储对象分配到 SPM 上时，只要这些存储对象不会在同一时间段内同时被使用，那么就可以将这样的存储对象分配到同一块 SPM 空间内。通过在代码中插入特殊的指令，可以实现存储对象在主存与 SPM 之间的换入换出，从而使得 SPM 在程序执行的不同时间段容纳不同的存储对象，充分利用程序的时间局部性。图 5-68 是可重叠优化方法的一个简单示例。

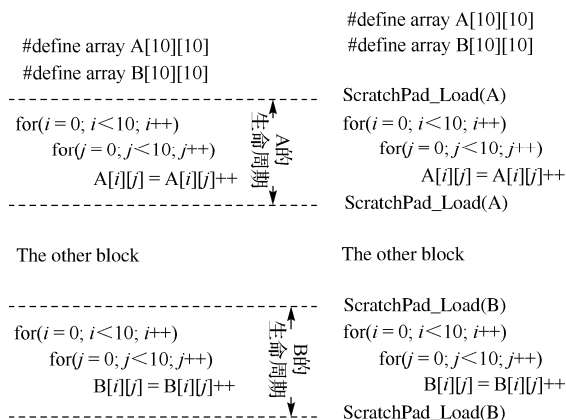


图 5-68 动态 SPM 优化技术示例

图 5-68 左侧的程序片段是没有优化时的代码。在这段代码中，对于数组 A 和数组 B 来说，它们的生命周期互不重叠，即它们不会在同一时刻被程序所使用。因此，可以将它们先后分配到 SPM 中，

以提高程序对数组 A 和数组 B 的访问速度。图 5-68 右侧是经过优化后的程序片段。在这段程序中，通过插入的 4 个函数，可以实现数组变量在 SPM 和主存之间动态换入换出。ScratchPad_Load (A) 表示从主存中读取数组 A 的数据到 SPM 中；ScratchPad_store (A) 表示将数组 A 的数据搬回主存。通过这种方式，当对数组 A 的使用结束后，将数组 A 搬回内存，在对数组 B 的使用开始时，通过 ScratchPad_Load (B) 指令将数组 B 搬运到 SPM，从而数组 A 和数组 B 可以重复使用 SPM。

由于片上存储器容量有限，传统的静态优化方法放入 SPM 中的存储对象内容有限，对于较大的程序，就不能像 Cache 一样更新程序热点的内容，因此，更多研究倾向于研究动态管理的 SPM。

3. SPM 和 Cache 共存架构的优化技术

SPM 的优化十分依赖于编译器或对程序 profile 信息的分析，而 Cache 则是完全对程序员透明的。由于造成 Cache 冲突的根本原因在于：处理器当前访问的数据被装入 Cache 时有很大概率将换出 Cache 对应行中已缓冲的、频繁访问的数据（具体的换出概率与 Cache 的组关联度与具体的换出策略有关），等到处理器下次访问原来的数据时，又不得不重新从片外存储器装载刚刚被换出的数据。这种情况造成的性能损失在程序具有较大的核心循环时尤为明显。

受到芯片面积、系统能耗的制约，嵌入式系统使用的 Cache 容量较小，一般采用单级架构；对于高端的，面向媒体计算的嵌入式系统中，也开始配置 2 级统一 Cache。因此对嵌入式系统的片上存储子系统中，关于 Cache 的研究趋向于对其参数的静态讨论与动态重配。

同为片上存储器，Cache 比 SPM 多了比较电路和 Tag 存储体，而 Cache 数据存储体部分和 SPM 一样都是由片上 SRAM 构成的，因此对于相同容量的 Cache 和 SPM 而言，SPM 在能耗、占用芯片面积两方面具有较大优势。但是基于 SPM 的优化方案需要软件管理，从而增加了程序管理的复杂性。我们认为 Cache 和 SPM 各具优势且存在互补性，因此两者共存的存储架构是将来嵌入式系统存储子系统发展的趋势。一些主流的处理器的内核，如 ARM926EJ-S、ARM1136、ARM1176-EJS 等均采用两者共存架构作为片上存储子系统。图 5-69 所示为 Cache 和 SPM 共存架构下的地址空间映射。

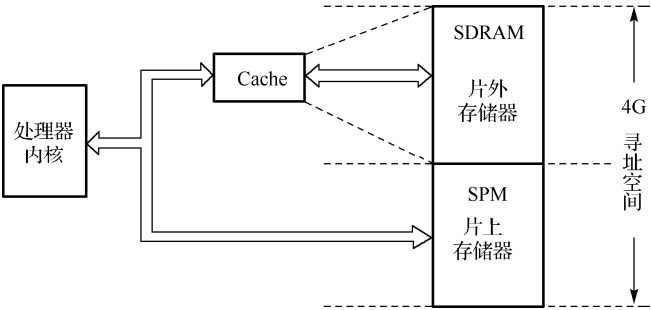


图 5-69 Cache 和 SPM 共存架构下的地址空间映射

以往的研究主要针对单纯配置 Cache 或单纯配置 SPM 的情况。但是，直接将仅针对 SPM 的优化算法与仅针对 Cache 的优化算法同时运用到两者共存的架构中，则可能将在一种存储体上取得的优化收益浪费到另一种存储器的开销上，甚至引入更多系统性能与能耗的额外开销。例如，针对 SPM 的优化算法将某段主存的地址空间搬运到 SPM，从而得到了性能与能耗的收益，然而搬运代码本身可能对指令 Cache 造成污染，引起 Cache 优化算法的失效，从而造成额外的 Cache 缺失，甚至导致 Cache 冲突。

目前针对 Cache/SPM 共存架构下的优化技术研究一般分为两类：

(1) 一些研究认为使用 SPM 比使用 Cache 更有优势，因此共存架构的优化算法主要通过对程序

的分析,将核心部分放入 SPM 以提升系统性能、降低系统能耗。而 Cache 则仅作为辅助存储,配置的容量较小,优化策略也比较简单。

(2)Cache 的冲突问题是造成系统性能降低、功耗提高的重要原因之一。一些研究将容易引起 Cache 冲突的程序段放入 SPM,从而减少了 Cache 缺失,也就减少了访问主存的次数,因此可以实现对性能、功耗的优化。

案例:高能效高清媒体处理器的访存 QoS^①

1. 市场驱动

对于当今的消费电子类产品而言,用户对于高清媒体的需求与期望越来越高。高级图形用户界面驱动着对于 GPU 的需求,另一方面,从手机到电视的各类产品对于高清视频的需求又驱动着对于视频加速的需求。随着 HDMI 高清视频接口在从机顶盒,手机甚至数码相机等几乎所有设备上的普及,1080p 全高清的图像与视频将成为几乎所有消费类电子产品的标配。在满足系统对于高清显示需求的同时保证成本和功耗的最优化是今天 SoC 设计者的巨大挑战。高清视频偶尔的不连续帧率,跳帧或者音频不连续,对于 PC 用户是可接受的,但对于高品质的消费电子产品而言却是无法容忍的。

2. 媒体处理器的内存架构——统一内存架构(UMA)与专用显存

对于媒体处理器而言存在两种基本的存储器架构:独立的显存架构与统一的内存架构。采用独立显存的系统中,图像和视频系统有自己独立的内存系统,CPU 则使用另一套存储系统。这就是典型的 PC 显存架构。采用这种架构的缺点第一是增加了系统成本,第二是使得内存管理变得更加复杂,因为系统中存在两套不同的存储系统。为了最大限度地提升系统的灵活性和降低成本,尤其是对于像智能手机这样工作负载变化巨大的系统,采用统一内存架构可能是更合适的选择。因此我们将重点讨论这种统一内存架构,虽然目前封装技术的进步已可以通过面对面(face-to-face)或芯片堆叠(chip on chip)技术将高带宽的存储器芯片封装在与处理器同一个封装中,从而实现消费电子产品的独立显存。

3. SDRAM 基础——行、列和 Bank

首先我们将简单回顾一下 SDRAM 的组织。通常 SDRAM 都包含若干个 Bank,每个 Bank 又包含了通过行列组织的存储体。地址被分为两个部分:行地址和列地址。在进行任何存储器访问之前,必须首先通过 Active 命令激活需要访问的行。对于已经被激活的行,可以通过列读取或者列写入命令对该行的内容进行读写操作。为了访问本 Bank 中的不同行,当前被激活的行需要首先被关闭(我们称之为预充电,Pre-charge),然后再通过行激活命令激活所需要访问的新行。在行打开命令与列读操作之间,以及列读操作到获得数据之间都存在一定的延迟。一般 SDRAM 芯片都包含 4 个或 8 个不同的 Bank,为了实现一定的并发性,这些 Bank 可以被独立地访问。因此,可以通过交织访问不同的 Bank 而隐藏行打开命令的延时。我们可以采用不同的地址映射机制来实现这种隐藏。比如,列地址不一定必须是处理器地址线的最低位。因为行打开操作是耗时和耗能的操作,SDRAM 控制器将尽可能调度内存访问发生在同一行中。对于处于同一行的访存命令,将具有更高的调度优先级。为了实现这种调度,SDRAM 控制器中通常都包含一个未完成交易队列,控制器可以乱序调度这些交易请求以实现行内访问的最大化。ARM 公司的 AMBA AXI 总线的每个交易都包含一个主设备 ID(Master ID),拥有

^① 本案例的内容摘译自 ARM 公司技术白皮书“QoS for High-Performance and Power-Efficient HD Multimedia”,作者:Ashley Stevens,出版时间:2010 年 4 月。

不同主设备 ID 的总线交易被认为是彼此独立的。另外一种方法是在 SDRAM 控制器上为每个主设备准备相应的独立 Slave 端口，这对于不支持乱序交易（Outstanding Transactions）的 AHB 总线而言是一种常用的方法。

4. 高清图像与视频的存储器带宽需求

图像与视频的带宽需求都取决于帧率、复杂度以及所采用的 GPU 和视频引擎。最普通的需求是帧率达到 30 帧每秒，因为在这个帧率下视频的显示才能够达到流畅。为了后续分析与讨论的方便，我们假设 GPU 在 1080p 30fps 帧率的条件下需要 1.5GB/s 的带宽，对于视频引擎，我们假设最坏情况下需要 500MB/s。另外对于 LCD 控制器而言，需要将 32bpp（每像素 32 比特）显示内容每秒 60 次地刷新到 LCD，这大概需要另外 500MB/s 的带宽。因此，对于一个分辨率 1080p、刷新频率为 60Hz 的系统而言通常需要 2GB/s 的存储器带宽。如果图形系统和视频同时显示的话，则需要 2.5GB/s。视频的后处理（Post-processing）还可能需更高的带宽。后面将以通常不需要视频后处理的移动设备（智能手机、平板电脑）为例，这类设备通常需要 2~2.5GB/s 的内存带宽。除了 GPU、视频引擎和 LCD 控制器对带宽的需求外，DMA 和 CPU 本身还要占用一部分内存带宽。

5. 可选的内存：DDR2、LP-DDR2 和 DDR3

当前消费电子类产品可选择的 SDRAM 存储器主要有 DDR2（低成本）、LP-DDR2（低能耗）和 DDR3（高性能）。DDR3 的每比特成本 2010 年将下降到和 DDR2 差不多。对于手机类的应用，通常选择 LP-DDR2，而笔记本、平板电脑和机顶盒、数字电视类应用则在 DDR2 和 DDR3 之间选择。在真实的系统中，SDRAM 的效率将远比理论峰值带宽要低，在某些情况下甚至只有峰值带宽的一半。

6. 不同的主设备需求

对于主设备而言，存储子系统有两个重要的性能指标：延迟与带宽。不同的主设备对于这两个性能指标的要求也各不相同。

（1）CPU：延迟敏感型设备。

如果 CPU 发起了对存储器的读请求，那么在这个读请求完成前，CPU 几乎只能等待直到交易完成。所以我们说 CPU 是延迟敏感型的设备，也就是说其性能直接受到有效延迟的影响，即使 CPU 对外存带宽的需求相对较低（由于受到 Cache 的影响）。这里所说的有效延迟是指总线主设备在该系统中所“看到”的仿存延迟，而不是存储器控制以及存储器颗粒的物理延迟。因为总线总裁以及其他总线主设备的竞争，有效延迟会比这个物理延迟大。有效访存延迟的变化对于处理器执行时间的影响参见图 5-70。

（2）GPU 与 VPU：带宽敏感型设备。

与 CPU 不同，GPU 需要很高的存储器带宽，而通常对仿存延迟不敏感。造成这种现象的原因是因为 GPU 总是对大量的独立数据进行相同或相似的处理，比如渲染时是对像素进行处理，而几何处理的时候则是对大量的三角形进行运算。虽然 GPU 需要进行大量的计算，但是这些计算并不依赖之前的计算结果。它不要停下并等待前面像素或三角形的运算结果。如果前面处理因为仿存原因而造成停顿，GPU 可以先计算后面的像素或三角形。VPU 对于视频数据的处理类似于 GPU，需要比较高的带宽（虽然没有 GPU 那么高），由于所处理数据之间的相关性比较低，VPU 对延迟的敏感度也比较低。总的来说，只要在一个给定的长时间（这里的长时间指的是长于一帧的时间）内保持足够的存储带宽，那么单次访问的延迟并不重要。

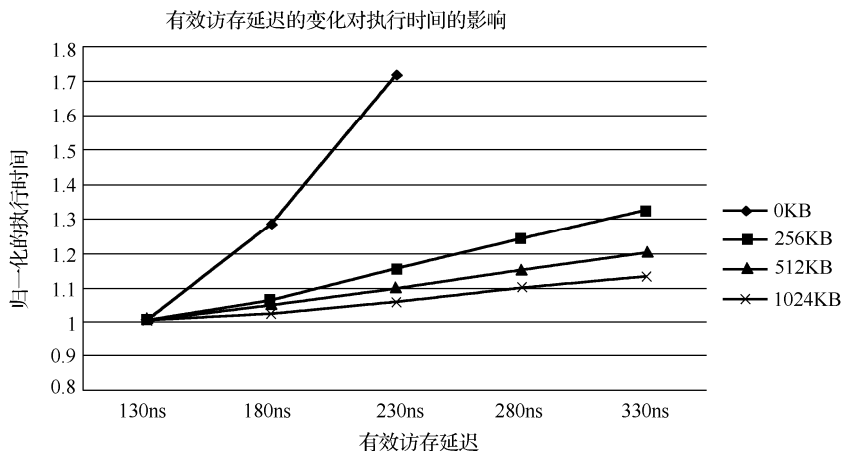


图 5-70 访存延迟的变化对执行时间的影响（数据来自 Cortex A8 在 Linux 上运行 Nightawk 浏览器）

（3）视频输出（LCD 或 HDMI 输出）：延迟关键型设备。

视频输出设备周期性地线性扫描帧缓冲（从低地址到高地址连续读取），数据必须按照固定的速率输出到显示设备，这时仿存的延迟是关键，如果显示设备中的数据 FIFO 发生了下溢，显示在屏幕上的画面将发生偏屏或裂屏，这将是灾难性的失败。因此，仿存延迟对于视频显示是非常关键的，虽然视频显示的带宽需求与 GPU 和 VPU 相比只能算是中等（未经压缩的 1080p 60Hz 刷新，32bpp 的显示输出需要 500MB/s 的带宽）。因为视频显示的地址模式是非常有规律的（通常就是线性递增），可以通过增大 FIFO 缓冲区，或者加大单次促发（Burst）读的长度等手段来缓解显示输出的延迟敏感性。但不管 FIFO 的大小是多少，对于每次交易的延迟都有相应的要求，而对于长期的平均延迟而言则必须满足帧时的需要。对于延迟敏感型的 CPU 而言，其性能直接受到仿存延迟的影响，也就是说仿存延迟越大，性能越低。但对延迟关键型的设备而言，有一个可接受的最大延迟，如果这个最大延迟不能满足则会发生灾难性的后果（比如裂屏），但是在满足这个最大延迟的前提下，进一步降低延迟对于其性能则没有影响。

7. SDRAM 控制器的调度效果

通过 SDRAM 控制器的调度机制将行内访问的优先级提高的办法，可以最大化 SDRAM 的效率。调度机制利用 SDRAM 控制器内部的等待交易队列进行仿存优化，目标是将行打开与行关闭的操作减到最低，这将减少在行打开与行关闭操作上所浪费的时间，同时降低由此而带来的额外能耗。我们定义 SDRAM 的带宽效率为实际的仿存带宽与理论带宽的比值。事实上，在实际应用中永远也不可能达到理论峰值带宽，因为毕竟 SDRAM 还必须有刷新时间，但我们可以通过调度有效提升带宽效率到 80%~85%，甚至更高。但是在我们努力提高带宽效率的同时，可能会引入未曾预料到的后果。一个延迟不敏感但对高带宽有需求的设备，可能会发出大量的 Outstanding 交易，产生大量的行内访问，从而使得延迟敏感和延迟关键型的设备的仿存延迟变大，从而进一步造成严重的系统性能问题。

所谓 Outstanding 交易，是指主设备在前一个仿存交易未完成的情况下发出一个新的仿存交易。通常认为 Outstanding 交易越多越好，因为这可能在很大程度上隐藏 SDRAM 的仿存延迟。SDRAM 控制器中包含了一个 Outstanding 交易队列用于暂存所接收的仿存请求，比如 ARM 公司的 DMC-341 和 DMC-342 两款 SDRAM 控制器可以被配置为支持 16 级 Outstanding 交易，实际应用中由于时序收敛的原因，通常被实现为支持 12 级。但某些主设备，比如 GPU，可以连续发出远多于队列长度的交易请求。这就造成了从片上互联（Interconnect）发往 SDRAM 控制器内部交易队列的通路被塞满了，其

他高优先级的 QoS 设备就不能向 SDRAM 控制器提交任何仿存交易请求，因为片上互联的仲裁器已经被锁死了。此时，SDRAM 控制器的 QoS 机制将彻底失效。设计人员能做的是如何避免这种极端情况的发生，或将其发生的概率降到最低。为了达到这个目的，我们必须确保 SDRAM 控制器中的交易队列总有空余的位置，并且片上互联支持 QoS 高优先级交易。这意味着限制高吞吐设备发出的 Outstanding 交易的速度，我们称其为“调控”（Regulation）。

8. 排队论与 Little 定律

排队论是研究队列问题的数学分支，其中著名的 Little 定律可以表示为：

$$N_T = R_T \times L_T$$

其中， N_T 表示在队列中的长时间平均客户数， R_T 则表示长时间平均客户到达率， L_T 表示客户在队列中的逗留时间。对于分析 SDRAM 控制器的讨论， N_T 表示队列长度， R_T 是交易到达率， L_T 是交易延迟。因此也可以将公式变为：

$$\text{队列长度} = \text{交易到达率} \times \text{交易延迟}$$

如果想限制队列的长度，可以直接将可能造成队列超出规定长度的非延迟敏感型交易请求延后，或者可以通过调节交易到达率和交易延迟来影响队列长度。

一种限制延迟的方法是采用远远超出需求的仿存带宽（比如 3~4 倍于所需带宽）。虽然这种方法对于简单、低端的微控制器系统而言简单易行（毕竟在这样的系统中，CPU 和其他总线设备的性能都非常有限），但对于现代高性能多媒体处理器而言却很难找到经济的解决方案。前面我们已经看到对于支持 1080p 的系统而言，仅仅支持图像显示就可能需要 1.75~2.35GB/s 的存储带宽，如果考虑其他主设备的需求，这个带宽可能要超过 2.5GB/s。而对于通常的低成本低功耗存储解决方案，比如采用 400Mbps、32 位 LP-DDR2 而言，其理论峰值带宽只有 3.2GB/s；32 位 DDR2 则可以提供理论峰值带宽 4.26GB/s，32 位 DDR3（中等成本与能耗）为 5.33GB/s。为了避免采用 64 位的存储系统（或者双通道的 32 位系统），我们需要系统的带宽效率达到 DDR3 的 50% 或者 LP-DDR2 的 80%，同时还必须保证延迟敏感型和延迟关键型的低延迟需求和带宽敏感型设备的带宽需求。为了实现这个设计目标，需要引入服务质量（Quality of Service, QoS）的机制。

（1）对 Outstanding 交易的调控。

经过前面的讨论我们已经知道需要对系统中的 Outstanding 交易的数量进行调控，以保证系统互联不会被太多的 Outstanding 交易塞满。最简单的做法当然是直接限制 Outstanding 交易的数量，比如对每个主设备分别规定其能发出的读写 Outstanding 交易数的最大值。ARM 公司的 NIC-301 和 QoS-301 片上互联 IP 核可以通过可编程寄存器对交易最大数进行调控。这使得系统人员可以在系统设计时对这些参数进行调整，而不是在芯片设计阶段就确定为固定值。QoS-301 甚至允许系统设计人员设置采用分数形式的平均 Outstanding 交易限制。

（2）对交易到达率进行调控。

Little 定律告诉我们：队列长度 = 交易到达率 × 延迟，因此可以通过调节交易到达率来间接调整队列的长度。对上面的公式进行变换得到：交易到达率 = 队列长度（Outstanding 交易数）/ 延迟。对于给定的交易量，到达率正比于带宽，因此调控交易到达率就等价于调控主设备的带宽。在实际的芯片中，这个带宽只是近似值，因为通常交易量是不固定的。SDRAM 控制器的猝发长度（Burst size）通常是固定的，因此对于总线上出现的长猝发，在 SDRAM 接口会被划分为多个固定长度的短猝发交易。另外，有些主设备根据系统的状况还能动态改变交易的大小。

ARM 公司的 QoS-301 可以通过控制每个主设备的交易发射率来调控每个主设备的带宽。这对于

有确定的最高带宽的设备而言是一个不错的解决方案，比如我们可以限制存储器到存储器的 DMA 传输的最大占用带宽。但是对于有多种功能的设备而言，这种固定最大带宽的方法不够灵活。交易发射率调控有时也称为 TSPEC (Traffic SPECification, 流量定制)，这属于来自网络的 QoS 研究。

(3) 对延迟的调控。

我们已经讨论了通过调控 Outstanding 交易数量直接控制队列长度，和通过调控交易到达率（也就是主设备的交易发射率）来间接控制队列长度。另一个 Little 定律告诉我们的可调控对象是延迟。我们无法直接控制延迟，但可以通过动态优先级的方法影响延迟。为每个从片上互联进入 SDRAM 控制器的交易请求赋予一个初始优先级，并通过动态调整这个优先级来达到一个长期的平均延迟。

9. 案例分析

本案例研究的硬件架构为：Cortex A9 双核 + PL310 二级缓存，高清 GPU 和视频引擎 (VE)。为了加快仿真速度，CPU、GPU 和 VE 都被替换成为 VPE (Verification and Performance Exploration, 验证与性能探索) 主设备模型，该模型是 ARM 用于在 RTL 仿真过程中生成或接受响应激励的模块。设计人员可以配置 VPE 模块发出符合上述主设备访存请求特征的激励，但仿真速度却要比真实的 CPU、GPU 和 VE 快得多。

GPU 和视频系统通常都采用双缓冲机制，一个帧缓冲用于显示，而另一个（也称为后台帧缓冲）用于 GPU 或视频引擎渲染输出。在 GPU 完成向后台帧缓冲的渲染输出后，系统将在场同步信号 (Vsync) 有效后切换前后台帧缓冲。在等待场同步信号的时间内，GPU 处于空闲状态并且可以处于关闭状态以节省功耗。如果 GPU 渲染的时间超过了一个帧，系统将重复显示前一帧直到 GPU 完成当前帧的渲染。因此 GPU 的访存特性表现为一段密集的访存请求（渲染时）和一段空闲期（等待场同步信号）。在本研究中，我们假设 GPU 渲染每一帧需要 50MB 数据，这样 30 帧每秒需要 1500MB 数据。因为 LCD 的刷新率是 60Hz，因此 GPU 渲染的每一帧 LCD 控制器将显示两遍。为了满足 30fps 的需求，GPU 每帧的渲染时间应该小于 33.3ms。

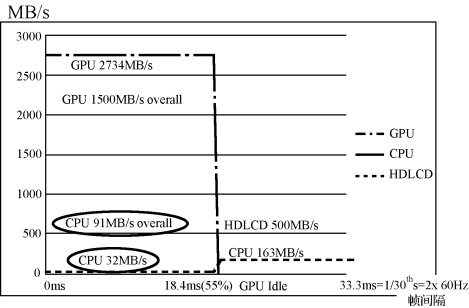
本案例对 3 种情况进行了研究：①对所有的总线主设备不做任何调控；②对 Outstanding 交易数进行调控，GPU 被限制为 3 读 1 写；③对交易发射率进行调控 (TSPEC)，GPU 被限制为 2.4GB/s 的带宽。3 种情况的仿真结果如图 5-71 所示。从图 5-71(a)中可以看到，GPU 在渲染期间（占整个帧时的 55%）占用了 2734MB/s 的带宽，由于 GPU 占用了太多带宽，在 GPU 活动期间，CPU 只能得到 32MB/s 的带宽，整个帧时的平均带宽仅为 91MB/s。在图 5-71(b)中，GPU 的 Outstanding 交易被限制为 3 读 1 写，从而导致其活跃期的带宽占用为 2664MB/s，下降了 3%。由于带宽占用的下降，GPU 的活动时间略微增加到了整个帧时的 56%。然而这点小小的变化却极大地改善了 CPU 的访存带宽，GPU 活动期间的 CPU 带宽提升到了 99MB/s，整个帧时的平均带宽也提升到了 127MB/s。这种 CPU 带宽的提升直接表现为 CPU 性能的提升，而 GPU 依然可以在一个帧时内完成渲染工作，因此 GPU 的性能并未受到影响。在图 5-71(c)中，GPU 的交易发射率受到调控，因为我们限制其带宽在 2441MB/s，渲染时间延长至整个帧时的 61%。CPU 在 GPU 活动期间的带宽上升到 119MB/s，整个帧时的平均带宽则上升到 136MB/s，从而在不牺牲 GPU 性能（因为 GPU 仍然可以按时完成渲染任务）的前提下，将 CPU 的性能大大提升。

图 5-72 给出了不同 QoS 策略下各总线主设备的平均读延迟和平均交易队列长度。可以看出由于采用了相应的调控措施，各个主设备的平均读延迟都有非常明显的下降，而交易队列的长度也相应地下降，其中 GPU 的交易队列长度下降非常明显。

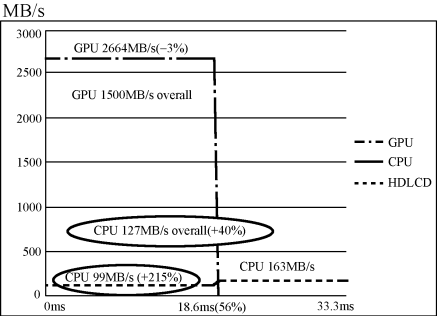
10. 小结

QoS 的目标是通过对 Outstanding 交易数直接或间接的调控，防止过多的此类交易造成片上互联的

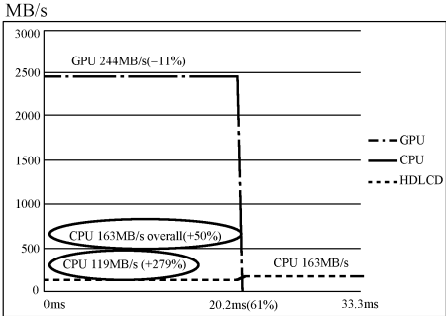
拥塞。然而，QoS 策略的使用必须节制，因为过分严格的调控策略可能造成 SDRAM 交易队列过短而无法隐藏 SDRAM 的延迟，从而降低整个 SDRAM 系统的效率。因此，需要对 QoS 策略进行小心而谨慎地微调，使得交易队列的长度足以隐藏 SDRAM 的延迟，同时又不至于造成队列过长而使得访存延迟变大。试图在硬件设计阶段完成这样的微调是非常困难的，因此通过软件在硅后（Post-Silicon）进行调整与优化就显得非常有必要了。这样做的另一个好处是可以根据不同的应用场景调整 SoC 的相关参数，以使其更佳地运行该应用。



(a) 不做任何限制与调控

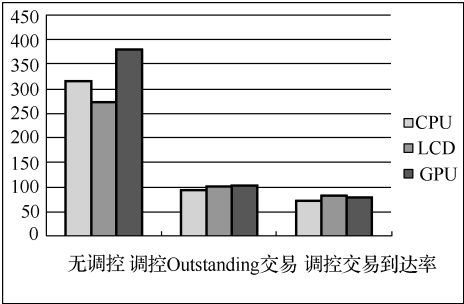


(b) 调控Outstanding交易数

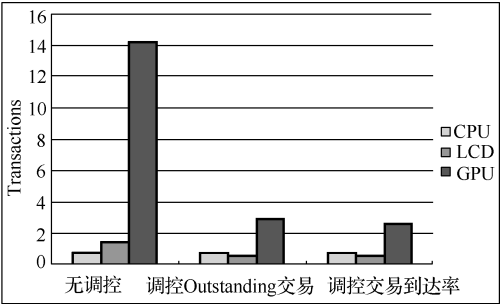


(c) 调控交易到达率

图 5-71 CPU、GPU、LCDC 在不同调控情况的带宽情况



(a) 不同QoS策略下的平均读延迟(周期)



(b) 平均交易队列长度

图 5-72 不同 QoS 策略下的平均读延迟与平均交易队列长度

思考题

1. SDRAM 和 DDR 存储器一般都采用行列地址分离的访存方式，即存储控制器首先在地址总线上输出行地址，然后再输出列地址，通过两个步骤完成地址的传输。这样的访存方式首先是由 SDRAM 的内部结构决定的（比

如需要首先行激活, 然后才能选择所要访问的字, 也就是列)。请问: 除此之外, 这种行列地址分离的设计还有什么样的优势? 为什么随着单颗存储器芯片容量的不断增加, 这样的设计优势很重要?

2. 本章 5.2.2 节中所介绍的 8 位地址直接映射 Cache 中, 对于一个索引地址为 ‘0b100’ 的 Cache 行, 其所映射的 4 个内存行地址分别是什么?

3. 本章 5.2.2 节中所介绍的 8 位地址四路组关联映射 Cache 中, 如果行大小是 8 字节, 请问其行内偏移地址占几位? 索引地址占几位? Tag 地址占几位?

4. 本章 5.2.6 节中所介绍的 ARM Cortex A8 处理器的 Cache 结构中, 其 L1 Cache 的容量是多少? L2 Cache 的容量又是多少?

5. 对于采用两级 Cache 的处理器而言, 就 L1 Cache 的设计来说是更重视访问延时还是命中率? 对于 L2 Cache 而言呢? 为什么?

6. 对于采用 32 位地址的系统而言, 其地址空间是 4GB, 如果每个内存页的大小为 4KB, 请问一个进程的页表大小是什么? 对于同样的系统, 32 个入口的 TLB 占用多少存储空间? (不考虑页表和 TLB 中的控制位)

7. 按照图 5-48 所给出的 NAND Flash 存储器的读时序, 为什么要将 NAND Flash 的行列地址分别按照 2 次和 3 次的方式进行传输? 按照这样的标准一片 NAND Flash 存储器理论上能够拥有的最大容量是多少?

8. 按照本章 5.5.3 节中所介绍的 OMAP 4460 处理器的存储空间分配方案, 讨论该芯片采用两个 DRAM 控制器 EMIF1 和 EMIF2 两个物理通道的设计有什么好处? 如果两个片选之间不进行地址交织, 这个好处还存在吗? 讨论交织粒度对于访存性能存在的潜在影响。

扩展阅读

- [1] Bruce Jacob, Spencer Ng, David Wang. Memory Systems: Cache, DRAM and Disk[M]. Morgan Kaufmann, 2007.
- [2] ARM. Cortex-A Series Programmer's Guide version 2. 0[EB/OL]. http://www.csc.lsu.edu/~whaley/teach/FHPO_F11/ARM/CortAProgGuide.Pdf.
- [3] SJE Wilton, NP Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model[J]. IEEE Journal of Solid State Circuits, 1996, 31 (5) :677-688.
- [4] Andrew S. Tanenbaum. Modern Operating System, 4th Edition[M]. Prentice Hall Press Upper Saddle River, NJ, USA. 2014.
- [5] 凌明. Cache/SPM 共存架构的动态存储布局优化技术研究[D]. 东南大学博士论文, 2011.
- [6] 王学香. SoC 高层建模和存储子系统内存布局优化技术研究[D]. 东南大学博士论文, 2009.
- [7] 浦汉来. SoC 存储子系统系统级性能优化技术研究[D]. 东南大学博士论文, 2006.
- [8] 曹飞. 针对 SEP0718 DDR 多端口存储控制器的调度算法的优化设计. 东南大学硕士论文, 2012.
- [9] Mutlu, Onur, and Lavanya Subramanian. Research problems and opportunities in memory systems[J]. Supercomputing Frontiers and Innovations 1, no. 3 (2014) : 19.
- [10] Hyungmin Cho, Bernhard Egger, Jaejin Lee, et al. Dynamic Data Scratchpad Memory Management for a Memory Subsystem with an MMU[C]. In Proceedings of the 2007 ACM conference on Languages, compilers, and tools for embedded systems. 2007. 195-206.
- [11] Milenkovic A, Milenkovic M, Barnes N. A performance evaluation of memory hierarchy in embedded systems[C]. Proceedings of the 35th Southeastern Symposium on System Theory, 2003. 427-431.
- [12] PR Panda, ND Dutt, A Nicolau. Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications[C]. Proceedings of the 1997 European conference on Design and Test, 1997. 7-11.
- [13] Avissar O, Barua R, Stewart D. An Optimal Memory Allocation Scheme for Scratch-Pad Based Embedded Systems

- [J]. ACM Transactions on Embedded Computing Systems (TECS), 2002, 1 (1): 6-26.
- [14] EDEC Solid State Technology Association. DOUBLE DATA RATE SDRAM SPECIFICATION [DB/OL]. <http://www.jedec.org/download/search/JESD79F.pdf>, 2008-02.
- [15] Elpida Memory, Inc. How to use DDR SDRAM[DB/OL]. <http://www.elpida.com/pdfs/E0234E40.pdf>, 2002-04.
- [16] Micron Technology, Inc. GENERAL DDR SDRAM FUNCTIONALITY [DB/OL]. <http://download.micron.com/pdf/technotes/TN4605.pdf>, 2001-07.
- [17] XILINX. LogiCORE IP Multi-Port Memory Controller (MPMC) [EB/OL]. http://www.xilinx.com/support/documentation/ip_documentation/mpmc/v6_04_a/mpmc.pdf, 2011.
- [18] K. L. E. Law. The bandwidth guaranteed prioritized queuing and its implementation[C]. In: IEEE Global Telecommunications Conference, 1997. 1445-1449.
- [19] W. -C. Kwon, S. Yoo, S. -M. Hong, et al. A practical approach of memory access parallelization to exploit multiple off-chip DDR memories[C]. in: Design Automation Conference, 2008. 447-452.
- [20] Zefu Dai, Mark Jarvin, Jianwen Zhu. Credit Borrow and Repay: Sharing DRAM with Minimum Latency and Bandwidth Guarantees[C]. In: IEEE/ACM International Conference on Computer-Aided Design, 2010. 197-204.
- [21] J. Corbal, R. Espasa, M. Valero. Command vector memory systems: High performance at low cost[C]. In: 1998 International Conference on Parallel Architectures and Compilation Techniques, 1998. 68-77.
- [22] Ye-Jyun Lin, Chia-Lin Yang, Tay-Jyi Lin, et al. Hierarchical Memory Scheduling for Multimedia MPSoCs[C]. In: IEEE/ACM International Conference on Computer-Aided Design, 2010. 190-196.
- [23] J. Shao, B. T. Davis. A burst scheduling access reordering mechanism[C]. In: IEEE 13th International Symposium on High Performance Computer Architecture, 2007. 285-294.
- [24] R. Crisp. Direct Rambus Technology: The New Main Memory Standard[J]. IEEE Micro, 1997, 17 (6): 18-28.
- [25] K. B. Lee, T. C. Lin, C. W. Jen. An efficient quality-aware memory controller for multimedia platform SoC[J]. IEEE Transactions on Circuits and Systems for Video Technology, 2005, 15 (5): 620-633.
- [26] Ibrahim Hur. Adaptive history-based memory schedulers[C]. In: 37th International Symposium on Microarchitecture, 2004. 343-354.
- [27] S. Whitty, R. Ernst. A bandwidth optimized SDRAM controller for the MORPHEUS reconfigurable architecture[C]. In: IEEE International Symposium on Parallel and Distributed Processing, 2008. 1-8.
- [28] O. Mutlu, T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems[C]. In: ISCA, 2008. 63-74.
- [29] E. Ipek, O. Mutlu, J. Martinez, et al. Self Optimizing Memory Controllers: A Reinforcement Learning Approach[C]. In: ISCA, 2008. 39-50.
- [30] S. Rixner, W. J. Dally, U. J. Kapasi, et al. Memory access scheduling[C]. In: ISCA, 2000. 128-138.
- [31] H. Lee, G. Tyson, M. Farrens. Eager Writeback—A Technique for Improving Bandwidth Utilization [C]. In: IEEE/ACM International Symposium on Microarchitecture, 2000. 11-21.
- [32] Jeffrey Stuecheli, D. Kaseridis, D. Daly, et al. Coordinating DRAM and Last-Level-Cache Policies with the Virtual Write Queue [J]. IEEE Micro, 2011, 31 (1): 72-82.
- [33] A. Burchard, E. H. Nowacka, A. Chauhan. A real-time streaming memory controller[C]. In: Proc. DATE, 2005. 20-25.
- [34] N. Rafique, W. T. Lim, M. Thottethodi. Effective management of dram bandwidth in multicore processors[C]. In 16th International Conference on Parallel Architecture and Compilation Techniques, 2007. 245-258.
- [35] Yonggon Kim, Hyunseok Lee, J. Kim. An Alternative Memory Access Scheduling in Manycore Accelerators[C].

- In:International Conference on PACT, 2011. 195-196.
- [36] H. G. Cragon. Memory Systems and Pipelined Processors[M]. Jones and Bartlett, 1996. 35-42.
- [37] S. A. Moyer. Access ordering and effective memory bandwidth [D]:[PhD thesis]. University of Virginia, 1993.
- [38] K. J. Nesbit, N. Aggarwal, J. Laudon, et al. Fair queuing memory systems[C]. In:IEEE/ACM International Symposium on Microarchitecture, 2006. 208-222.
- [39] Micron Technology. 512MbDDR2[EB/OL]. <http://www.micron.com/parts/dram/ddr2-sdram/~ /media/Documents/Products/Data%20Sheet/DRAM/455512MbDDR2.ashx>, 2004.
- [40] JEDEC SOLID STATE TECHNOLOGY ASSOCIATION. DDR2 SDRAM SPECIFICATION [EB/OL]. <http://www.jedec.org/sites/default/files/docs/JESD79-2F.pdf>, 2009.
- [41] Xiao, L. , Zhang, X. , and Kubricht, S. A. Improving memory performance of sorting algorithms[J]. ACM Journal on Experimental Algorithmics, 2000. 23-28.
- [42] Pawlowski, J. Thomas. Hybrid memory cube (HMC) [C]. In Hot Chips, vol. 23. 2011.
- [43] Rosenfeld, Paul. Performance exploration of the hybrid memory cube [D]. university of Maryland, 2014.

第 6 章 外 设 接 口

本章介绍嵌入式系统中常用的一些外设接口的功能、SEP4020 中相应的外设接口控制器及其驱动。首先介绍低速通信接口，包括异步串行通信 UART、同步串行通信 I2C 和 SPI；接着介绍高速通信接口，包括通用串行总线 USB、10M/100M/以太网 MAC 网络接口；然后介绍人机接口，包括液晶显示器接口、音频接口和触摸屏接口；最后介绍嵌入式系统中的定时器。

6.1 低速通信接口

首先解释一下什么是串行通信和并行通信，以及什么是异步通信和同步通信。

串行通信是指计算机与 I/O 设备之间传输的数据是一位接一位依次传送的。通常数据在一根数据线上传输。串行通信的传输速度慢，但使用的传输设备成本低，可利用现有的通信手段和通信设备，适合于计算机的远程通信。

并行通信是指计算机与 I/O 口设备间通过多条数据传输线交换数据，数据的各位同时进行传送。并行通信的速度快，但使用的传输设备成本高，适合于近距离的数据传输。

异步通信是指数据传送以字符为单位；用起始位和停止位标识每个字符的开始和结束；相邻两字符之间间隔任意长；因为一个字符中的比特位长度有限，所以只要接收时钟频率和发送时钟频率相同就可以。异步通信不需要发送方和接收方之间传送同步时钟信号（也就是在接收方和发送方之间没有时钟线）。

同步通信以数据块为单位传送信息。在一个数据块（信息帧）内，字符与字符间无间隔；在数据块开始处要用同步字符来指示，因为一次传输的数据块中包含的数据较多，所以接收时钟与发送时钟严格同步，通常要有同步时钟（该同步时钟信号由通信双方的一方，可以是接收方，也可以是发送方，发送给另一方用于采样时钟）。

串行通信可以是同步的，也可以是异步的。并行通信由于速度较高，一般都采用同步方式进行。

6.1.1 异步串行通信 UART

1. 异步串行通信 UART 概述

通用异步接收器和发射器（UART，Universal Asynchronous Receiver/Transmitter）是一种通用串行数据传输协议，可以实现全双工异步通信。用于同串行输入和串行输出的装置进行通信，串行传输以速度为代价，换取了成本和连线复杂程度的降低，对于许多应用而言，这是一个令人满意的结果。串行数据流的同步是通过给发送数据增加起始位和停止位以形成一个数据字符而实现的，数据完整性是通过在数据字符中附加一个奇偶位来实现的，由接收器来检验此奇偶位是否有数据位出错。对于主机系统，UART 就像一个能读取和写入的 8 位输入和输出端口，任何时候，对于主机要发送的数据，它只需以字节格式把这些数据发送到 UART（8 比特位宽），当 UART 从另一个串行装置接收数据时，它把这些数据临时缓存在它的 FIFO 中（同样是 8 比特位宽），然后通过一个内部寄存器位或通过一个硬件中断信号通知 CPU 可以使用这些数据。其工作原理是将传输数据的每个字符分别按位传输。其中各位的意义如下：

- 起始位：先发出一个逻辑“0”的信号，表示传输字符的开始。
- 数据位：紧接着起始位之后。数据位的个数可以是4、5、6、7、8等，构成一个字符。通常采用ASCII码，从最低位开始传送。
- 奇偶校验位：数据位加上这一位后，使得“1”的位数应为偶数（偶校验）或奇数（奇校验），以此来校验数据传送的正确性。
- 停止位：是一个字符数据的结束标志。可以是1位、1.5位、2位的高电平。由于数据传输是以异步形式进行的，每一个设备有其自己的时钟，在通信中两台设备间很可能出现小小的不同步。因此停止位不仅仅表示传输的结束，还提供设备间进行时钟同步的机会。但是停止位的位数越多，数据传输率同时也越慢。
- 空闲位：处于逻辑“1”状态，表示当前线路上没有数据传送。

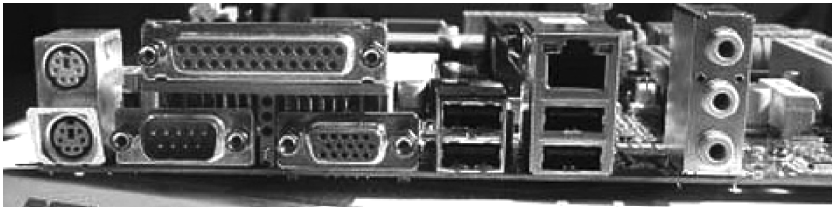
UART 传输的发送器按位发送数据，每一位时间固定，接收端以相同的时间间隔采样数据。该时间反映了 UART 传输最重要的参数，即波特率（**baud rate**）。在通信原理中波特率是指数据信号对载波的调制速率。对于 UART 传输，处理的都是数字信号，并没有调制信号，波特率表示每秒传送的二进制位数，是衡量数据传送速率的指标。

电子工业联合会（EIA, Electronic Industries Association）定义了若干种 UART 传输接口规范，如 RS-232、RS-422 及 RS-485 等。不同规范的信号数量、传输速率、电气特性、接口机械特性各不相同，并对基本的 UART 传输协议做了相应的补充。如 RS-232 支持硬件流控，规范定义了相应的信号线及使用规范；RS232 只能连接一对设备，而 RS-422 及 RS-485 则可以连接多个设备，规范定义了不同设备间传输信息的内容及格式，包括主机轮询格式、主机的编码方法、数据格式等。

RS-232-C 标准是美国 EIA（电子工业联合会）与 BELL 等公司一起开发的、于 1969 年公布的通信协议。RS 是英文 **Recommended Standard**（推荐标准）的缩写，232 为标识号，C 表示修改次数，以下简称 RS232。RS232 适合于数据传输速率在 0~20 000bps 范围内的通信。这个标准对串行通信接口的相关问题，如信号线功能、电气特性都做了明确规定。由于通信设备厂商都生产与 RS232 制式兼容的通信设备，因此，它作为一种标准，目前已在微机通信接口中广泛采用。一般计算机上都带有 RS232 接口，如图 6-1 所示，称为串口（COM）。

- 通常 RS232 接口以 9 个引脚（DB-9）或是 25 个引脚（DB-25）的型态出现（DB-25 接口目前已极少使用）。对于一般双工通信，仅需 3 条信号线即可实现，一条发送线、一条接收线及一条地线。
- RS232 标准规定的数据传输速率，即波特率为 50bps、75bps、100bps、150bps、300bps、600bps、1200bps、2400bps、4800bps、9600bps、19 200bps、38 400bps。
- RS232 标准规定，驱动器允许有 2500pF 的电容负载，通信距离将受此电容限制。若每米电缆的电容量减小，通信距离可以增加。传输距离短的另一个原因是 RS-232 存在共地噪声和不能抑制共模干扰等问题，因此一般用于 20m 以内的通信。
- RS232 标准对逻辑电平进行了定义：逻辑“1”的电平为-3V~-15V，逻辑“0”的电平为+3V~+15V。也就是当传输电平的绝对值大于 3V 时，电路可以有效地检查出来，介于-3V~+3V 之间的电压无意义，低于-15V 或高于+15V 的电压也认为无意义，因此，实际工作时，应保证电平在±(3~15)V 之间。

RS232 是用正负电压来表示逻辑状态的，集成电路采用的 TTL（Transistor-Transistor Logic，逻辑门电路）器件是以高低电平表示逻辑状态的。因此，为了使用 RS232 接口进行 TTL 器件之间的数据传输，必须进行电平和逻辑关系的变换。实现这种变换的方法可用分立元件，也可用集成电路芯片。目前比较广泛地使用集成电路转换器件，如 MAX232 芯片可完成双向电平转换。



外形	引脚	符号	输入/输出	说明
	1	DCD	输入	数据载波检测
	2	RXD	输入	接收数据
	3	TXD	输出	发送数据
	4	DTR	输出	数据终端准备好
	5	GND	—	信号地
	6	DSR	输入	数据装置准备好
	7	RTS	输出	请求发送
	8	CTS	输入	允许发送
	9	RI	输入	振铃指示

图 6-1 RS232 DB-9 接口

RS232 接口的串行接口信号如下：

- TXD——发送数据信号。在正常工作模式下，其以一定的波特率发送串行数据。负电平表示逻辑“1”，正电平表示逻辑“0”。在红外模式下，TXD 信号发送经过处理的脉冲信号，窄的正脉冲表示逻辑“0”，逻辑“1”则保持低电平。
- RXD——接收数据信号。在正常工作模式下，其接收来自外部的串行数据。在红外模式下，窄的负脉冲表示逻辑“1”，逻辑“0”则保持高电平。
- $\overline{\text{CTS}}$ ——The Clear To Send signal。允许发送信号，低电平有效。当此信号有效时，表示外部设备可以接收 UART 发送的数据。
- $\overline{\text{RTS}}$ ——The Request To Send signal。请求发送信号，低电平有效。UART 向外部设备发出的数据请求信号。外设只有探测到此信号有效时才可以发送数据。
- $\overline{\text{DTR}}$ ——The Data Terminal Ready signal。数据终端（DTE）准备好信号，低电平有效。一般而言，只表示设备可用。可以通过 MCR 寄存器的第 0 比特设置。
- $\overline{\text{DSR}}$ ——The Data Set Ready signal。数据设备准备好信号，低电平有效。表示数据设备可用。是 $\overline{\text{DTR}}$ 信号的应答信号。
- $\overline{\text{DCD}}$ ——The Data Carrier Detect signal。接收线路载波检测信号，低电平有效。通知 UART 探测到载波，准备接收数据。
- RI——The Ring Indicator。振铃指示信号，低电平有效。数据设备（DCE）通知 UART 接收到一个电话振铃。
- out_baud——The Baud Clock signal。这是 UART 双方共同约定使用的、用于传输串行数据的时钟。
- $\overline{\text{out1}}$ ——可控制输出信号 1。由 MCR 寄存器第 2 比特控制。
- $\overline{\text{out2}}$ ——可控制输出信号 2。由 MCR 寄存器第 3 比特控制。

最简单的串行通信仅需要 3 条信号线：发送信号线（Tx）、接收信号线（Rx）及地线（作为其他两条信号线的电平参考）。将一个设备的 Tx 信号线与另一个设备的 Rx 信号线相连即可实现通信。通过增加硬件流控信号，可以提高串行通信的效率。此外，过去广泛使用的调制解调器（现在已极少使用）多使用串行通信接口与计算机相连，接口还带有反映调制解调器状态的信号线。

UART 传输可以增加相应的硬件流控功能，以提高传输效率。RS232 规范就定义了相关标准。流

控 (flow control) 用于表示接收器暂时无法接收数据, 发送器需停止传输。流控有两种形式: 软件流控与硬件流控。对于 RS232 规范来说硬件流控使用两条额外的信号线来实现流控功能: $\overline{\text{RTS}}$ (Request to Send) 以及 $\overline{\text{CTS}}$ (Clear to Send)。将数据发送端的 $\overline{\text{CTS}}$ 线与数据接收端的 $\overline{\text{RTS}}$ 线相连接, 如果接收端准备停止接收数据, 就将 $\overline{\text{RTS}}$ 信号置为逻辑 0, 请求停止发送; 接收端可以接收更多数据时, 将 $\overline{\text{RTS}}$ 信号线改为逻辑 1, 恢复数据传输。

硬件流控是靠软件实现的, 之所以强调“硬件”二字, 仅仅是因为硬件流控提供了用于流量情况指示的硬件信号线 ($\overline{\text{CTS}}/\overline{\text{RTS}}$)。并不是说, 只要把信号线连上, 硬件就能自行流控。如果软件不支持, 只连接上 $\overline{\text{RTS}}$ 和 $\overline{\text{CTS}}$ 是没有用的。因此, 通常情况下软件流控更常使用, 数据接收端向数据发送端发送特定的字符数据以请求发送端暂停数据传输, 再发送特定的字符数据以恢复数据传输。

在计算机平台上, 通常使用专用的串口芯片, 该芯片提供 RS232 接口供处理器使用。较为常见的串口芯片为 8250, 后改进为 16550A。8250 是 IBM PC 及兼容机使用的第一种串口芯片。8250 支持的最大波特率为 56kbps。16550A 的管脚与 8250 相同, 驱动软件也与 8250 兼容, 提供更高的性能。16550A 的最大波特率为 256kbps (由于硬件设计原因, 编程时最大只能使用 115200)。16550A 性能增强的关键是使用了 FIFO 寄存器作为数据缓冲 (16 字节的发送 FIFO 寄存器和 16 字节的接收 FIFO 寄存器)。

在嵌入式系统中, 串口芯片通常集成在 SoC 内, 即 UART 传输控制器。CPU 通过 UART 传输控制器与外设进行 UART 传输。将 CPU 传送过来的要发送的并行数据转换为串行数据流输出, 并将外设来的串行数据转换为并行数据, 供 CPU 使用。CPU 把准备发送的数据写入 UART 传输控制器的发送数据寄存器中 (可以是 FIFO 寄存器, 以提高传输性能); 相应的 CPU 可以从 UART 传输控制器的接收数据寄存器 (可以是 FIFO 寄存器, 以提高传输性能) 依次读出接收到的数据。

2. SEP4020 中的 UART 控制器

SEP4020 处理器中的 UART 控制器支持 RS-232 标准的非归零的编码格式, 也支持红外模式下的编码格式。它通过 RS-232 协议和外设进行串行通信。同时, 也可以通过外围电路将红外信号转变成电信号 (接收信息时) 或将电信号转变成红外信号 (发送时), 然后再和 UART 进行串行通信, 这样就可以支持低速红外通信。

UART 控制器收发字符位数是可配置的, 可以是 5~8 位。在发送时, 数据从数据总线上写入发送 FIFO, 然后再被送入移位寄存器, 转换成串行数据, 从 TXD 引脚输出。在接收时, 数据从 RXD 引脚串行地接收, 先送入接收移位寄存器, 然后再送入接收 FIFO。接收 FIFO 和发送 FIFO 都可发出可编程的边沿触发中断, 当 FIFO 中的数据量达到预先设定的数据量时, 可发出中断。同时, 接收 FIFO 和发送 FIFO 都可以请求进行 DMA 传送。

UART 控制器的传输波特率取决于输入时钟和可编程的分频器。UART 还具有波特率自检测单元电路, 这一电路单元可以配置停止位数目及奇/偶校验。在接收时, 可以检测帧错误、空闲状态、中止符、奇偶校验错误及溢出错误。

SEP4020 中 UART 控制器的特性:

- 全双工操作。
- 5~8 位字符操作。
- 可配置的奇偶校验 (偶校验, 奇校验, 不用奇偶校验, 或固定校验位)。
- 可配置的停止位 (1 位, 1.5 位, 2 位)。
- Break 产生和探测功能。
- 16 级深度 (字节宽度) 的接收 FIFO, 可配置触发级中断和超时中断。
- 16 级深度 (字节宽度) 的发送 FIFO, 可配置触发级中断。

- 对 $\overline{\text{RTS}}$ 、 $\overline{\text{CTS}}$ 信号提供硬件控制流支持。
- 提供 $\overline{\text{DSR}}$ 、 $\overline{\text{DCD}}$ 、 $\overline{\text{RI}}$ 、 $\overline{\text{DTR}}$ 通信信号。
- MODEM 状态中断。
- 4 位可屏蔽中断源，中断优先级处理，超时中断。
- 灵活的波特率配置。
- 用于高速同步通信的 clock x1 时钟输出。
- 支持串行红外接口物理层协议。

SEP4020 的 UART 控制器的功能框图如图 6-2 所示。

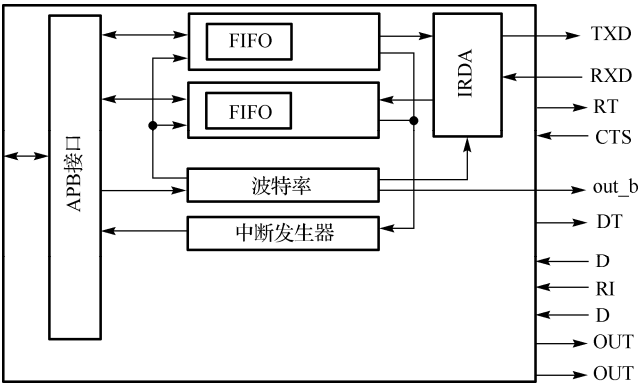


图 6-2 UART 控制器功能框图

1) UART 控制器发送数据的过程

发送器从 CPU 接收并行数据，存入发送 FIFO，然后在字符中加入起始位、奇偶校验位、停止位后按照设置的波特率串行（LSB first）发送出去。

发送器 Tx 具有一个可屏蔽的中断源：发送器可配置触发级中断。一个 16 级深度的 FIFO 可以根据系统的情况配置发送器请求数据的触发级。当发送 FIFO 中的数据小于等于配置的 FIFO 触发级时，产生中断（如果没有屏蔽中断源）。比如设置为半空中断时，如果 FIFO 中的数据少于 8 个，则触发中断。通过向发送 FIFO 中写数据可以清除中断。中断清除后只有先使发送 FIFO 中的数据大于某个值后，才能再次产生中断。当发送 FIFO 中数据已满时，再写入的数据将被丢弃。

$\overline{\text{CTS}}$ 用于硬件自动流控。如果使能硬件自动流控并且 $\overline{\text{CTS}}$ 为高（无效状态），则发送器在发送完当前的字符后进入等待状态，直到 $\overline{\text{CTS}}$ 变为低（有效状态）再接着发送字符。 $\overline{\text{CTS}}$ 的状态变化以及当前状态可以在 UART 控制器的状态寄存器中反映出来。

2) UART 控制器接收数据的过程

接收器 Rx 串行地接收数据，并把它们转变成并行的字符。当开始接收时，接收器检测起始位，接下来，在每个数据位的中间时刻采样该数据位以确定其逻辑状态。一旦起始位被确认，以后的数据位、奇偶校验位、停止位依次被接收。奇偶校验位将被检测并且在 LSR 寄存器会反映其状态。同样，帧错误、中止符等也都会被检测并报告其状态。

接收器 Rx 具有两个可屏蔽中断源和一个 timeout 中断源。两个可屏蔽中断源分别为：接收器可配置触发级中断、接收数据错误中断。一个 16 级深度 FIFO，可以根据系统的情况配置接收器产生中断的触发级。当接收 FIFO 中的数据大于等于配置的触发级时，产生中断（如果没有屏蔽中断源）。比如，配置为半满中断时，当接收 FIFO 中的数据大于 8 时，产生中断。通过读取接收 FIFO 中的数据至小于配置的触发级可以清除中断。当接收 FIFO 中数据已满时，接收器新接收的数据将被丢弃并产生 overrun

错误。如果 CPU 从接收 FIFO 中读取的当前数据有奇偶校验或帧错误，则 UART 控制器的状态寄存器的某一位被置位，同时产生中断（如果没有屏蔽中断源）。如果接收 FIFO 接收的数据中有奇偶校验、帧错误或接收器探测到 break，则 UART 控制器的状态寄存器的某一位被置位。如果接收 FIFO 不空并且在一定的时间内没有被读空，则产生 timeout 中断。

RTS 用于硬件自动流控。如果使能硬件自动流控，当接收 FIFO 达到配置触发级时，RTS 变为高（无效状态）以通知外部设备暂停发送。读空接收 FIFO 中的数据后，RTS 变为低（有效状态），外部设备可以继续发送数据。

3) 红外接口

红外接口将待传输的数据转变成 IRDA 串行红外物理层所规定的格式。

在发射时，如果是“0”则发出一个窄的正脉冲，其宽度为 3/16 个周期；如果是“1”则不发出脉冲，而外围电路则需驱动红外 LED。

在接收时，如果是“1”则输入一个窄的负脉冲；如果是“0”则不输入脉冲，从 RXD 引脚上接收的数据通过多路选择器送到相应的解码单元，外围电路将红外信号转变成电信号。

3. UART 驱动

首先给出 SEP4020 处理器中的 UART 模块提供给用户进行配置的寄存器：

- 中断使能寄存器。
- 中断识别寄存器。
- FIFO 控制寄存器。
- 线路控制寄存器。
- Modem 控制寄存器。
- 线路状态寄存器。
- Modem 状态寄存器。
- Divisor Latches 寄存器：用于设置传输的波特率。
- 接收 FIFO。
- 发送 FIFO。

UART 驱动包括下面几部分：

- (1) 初始化串口。清空接收 FIFO 和发送 FIFO，关闭相应中断，对有关寄存器进行初始化。
- (2) 配置串口。通过默认值配置串口，也可以通过配置设置用户需要的相关参数，可以设置串口、波特率、数据位、奇偶校验位和停止位。
- (3) 中断服务程序入口函数。通过产生的中断号进行判断，调用相应的中断服务程序。
- (4) 发送数据。当系统任务调用函数将要发送的数据复制到缓冲区中后，接着就打开发送中断，数据通过中断发送出去。通过设定两个全局指针来控制是否发送完毕。当最后一个中断发现已经没有要发送的数据后，关闭发送中断（但这个时候由于发送空中断已经被锁存进 UART 的相关寄存器中，所以直接在中断控制器中关闭了 UART 的中断）。
- (5) 接收数据。接收数据的操作放在中断处理程序中进行，将缓冲区中的数据写入用户指定的目标区，设定两个全局的指针控制是否接收完毕。由于接收有触发级中断和超时中断两个中断，因此驱动程序中包含两个接收中断服务程序，对于每一种情况均要判断是否接收完毕，从而关闭中断。

为了防止由于传送错误使得串口任务长期占有系统，在接收和发送数据中还设有 timeout 控制，对一定数量的数据传送限定了相应的时间限制，使传送在系统可以容忍的范围内进行。

注意：由于在接收和发送数据时需要关闭 UART 中断，导致在实际串口工作时只能以半双工方式运行，如果需要全双工方式运行，需要修改相关驱动程序中的中断设置。

图 6-3 是 UART 驱动程序的函数调用流程图。

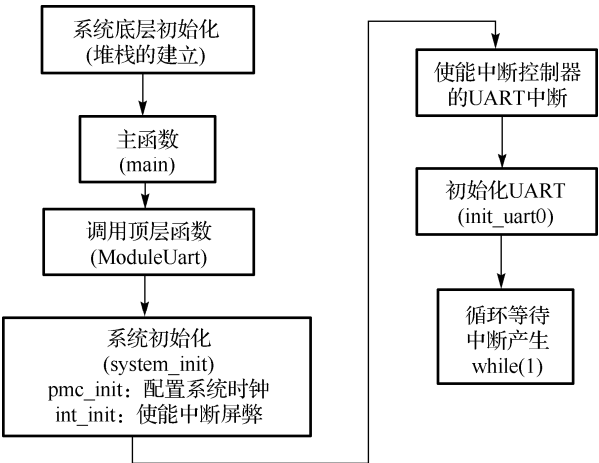


图 6-3 UART 驱动程序流程图

6.1.2 同步串行通信

1. I2C 总线

I2C（Inter-Integrated Circuit）总线是 Philips 公司在 1987 年获得的专利，必须得到 Philips 的授权才能使用，比如 Maxim 带有 I2C 接口功能的产品中一般都声明：“购买 Maxim Integrated Products, Inc. 或其他经过认证的相关公司的 I2C 产品，需转让将这些产品用于 I2C 系统的 Philips I2C 专利许可协议，保证系统符合 Philips 定义的 I2C 标准规范。”

I2C 协议虽然简单，却受到 Philips 专利的限制。ATMEL、Maxim、Cirrus Logic、Linear 等厂商都曾被 Philips 起诉侵权。很多厂商采用多种手段来规避 Philips 专利，最典型的就是避称 I2C，而采用其他称谓。一些厂家把串行控制接口设计成与 I2C 兼容（I2C 协议的子集），称为双线（two wire）接口，如今手机市场中应用最多的 CMOS Sensor 芯片的串行控制接口采用的就是这种策略。另一种策略则是采用软件操作通用输入/输出端口模拟 I2C 接口时序的策略与其他设备通信，而不将 I2C 协议固化在硬件中，此举也可有效避开 Philips 专利。

不过，基本的 I2C 专利在 2004 年已经期满，大多数 I2C 应用已经不受 Philips 的专利限制，从基本 I2C 扩展的一些高速规范以及 I2C 地址分配等仍然受到专利限制。

I2C 总线由主机与从机组成。主机用于初始化总线数据传输、产生时钟信号和终止数据传输。此时，任何被寻址的器件都被认为是从机。I2C 总线具有如下特征。

- （1）只要求两条总线线路：一条串行数据线 SDA，一条串行时钟线 SCL。两条线路均为双向线路，分别通过上拉电阻连接到电源。
- （2）真正的多主机总线：如果两个或更多主机同时初始化，数据传输可以通过冲突检测和仲裁防止数据被破坏；主机可以作为发送器（发送数据到总线）或接收器（从总线接收数据）。
- （3）每个连接到总线上的器件都可以存在唯一的地址。

(4) 串行的 8 位双向数据传输位速率在标准模式下可达 100kb/s，快速模式下可达 400kb/s，高速模式下可达 3.4Mb/s。

(5) 片上滤波器可以有效滤去总线数据线上的毛刺，保证数据正确。

(6) 连接到总线上的器件数量只受总线最大电容 400pF 的限制。

一般情况下，I2C 总线为单主机多从机结构。从机地址位为 7 位（后增加到 10 位，10 位寻址与 7 位寻址兼容，不会影响已有的 7 位寻址设备，可以连接到相同的 I2C 总线上。主机通过指定从机地址选择特定从机进行操作。通常情况下从机并不需要如此多的地址位数，因此通常地址高位由从机固化，地址低位通过从机管脚由用户指定，如图 6-4 所示。

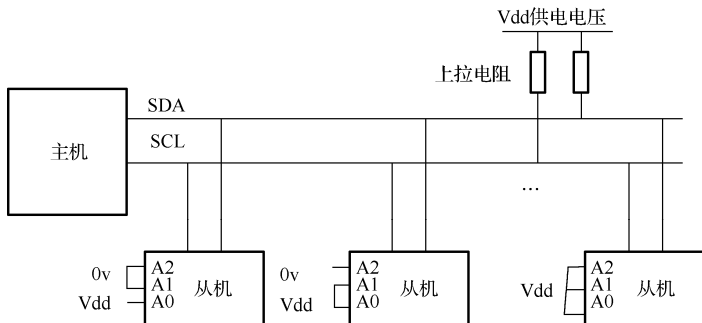


图 6-4 I2C 总线

I2C 总线采用两条双向线路直接连接总线上的各个器件。各个器件端口采用集电极开路设计，多个器件端口可以直接相连，通过上拉电阻连接电源。任意设备输出低电平时将使得整条线路变为低电平，所有设备均输出高电平（或没有输出）时，线路由上拉电阻设置为高电平。

I2C 总线设备存在 3 种工作模式：标准（SS，Standard Speed）模式、快速模式（FS，Fast Speed）及高速（HS，High Speed）模式。标准模式器件数据传输率可达 100kb/s。快速模式器件可以在 400kb/s 下接收和发送。快速模式器件向下兼容，可以和标准模式器件在 0~100kb/s 的 I2C 总线系统中进行通信。标准模式器件不向上兼容，所以不能在快速模式下工作。高速模式器件的传输速度有较大突破。高速模式器件可以在高达 3.4Mb/s 的位速率下传输信息，而且保持向下兼容快速模式或标准模式器件，它们可以在一个速度混合的总线系统中双向通信。高速模式传输除了不执行仲裁和时钟同步外，总线协议和数据格式与快速/标准模式传输相同。

在 I2C 总线传输中，主机用于初始化总线数据传输、产生时钟信号和终止数据传输。传输过程如下（以 7 位寻址高速/标准模式传输为例）。

(1) 开始传输。

SCL 线是高电平时，SDA 线从高电平向低电平切换，这个情况表示起始条件（S）。传输必须由起始条件开始，总线在起始条件后被认为处于忙的状态。

(2) 数据传输。

在传输数据的时候，SDA 线必须在 SCL 线高电平时保持稳定，SDA 线的电平状态只有在 SCL 线低电平时才能改变。发送到 SDA 线上的每个字节必须为 8 位，每次传输可以发送的字节数量不受限制。

传输的第一个字节头 7 位为从机地址，首先传输的是地址的最高位，第 8 位为读/写控制位，决定了传输的方向。当发送了一个地址后，系统中的每个器件都在起始条件后将头 7 位数据与自身地址进行比较，如果一样，器件会判定被主机寻址，至于是接收器还是发送器，由读/写控

制位决定。最低位为“0”，表示主机写信息到被选中的从机；“1”表示主机从被选中的从机读取信息。

主机发送器件寻址及读/写控制字节后，就可以进行数据传输了。数据以字节为单位传输（通常高位在前），可以连续进行多个字节数据传输。传输的数据格式由具体器件决定。例如，对于常用 I2C 串行接口 EEPROM 的读/写操作，向器件写入一个字节数据，需要在器件寻址及读/写控制字节后发送两个字节数据，第一个字节指明器件内要写的存储器地址，第二个字节即所写数据；从器件读出数据有两种形式：①器件设计有当前地址计数器，复位值为 0，每次读操作器件将返回地址计数器对应的存储器数据，同时地址计数器递增。②器件写操作（可以只发送存储器地址，不发送写入数据）后，当前地址计数器的值变为写操作指定的存储器地址，之后就可以对该存储器地址进行读取操作了。两种形式的读取操作都可以连续进行，即地址计数不断递增，返回数据。

如果从机要完成一些其他功能后才能接收或发送下一个数据字节，可以使 SCL 线保持低电平，迫使主机进入等待状态，当从机准备好接收下一个数据字节并释放 SCL 线后数据传输继续进行。

数据传输（包括地址位/控制位）必须带响应，每个字节后必须跟一个响应位。相关的响应时钟脉冲由主机产生。在响应的时钟脉冲期间发送器释放 SDA 线，SDA 线的状态由接收器决定。接收器拉低 SDA 线，在时钟脉冲的高电平期间保持稳定的低电平，表示接收器接收数据；接收器没有拉低 SDA 线，SDA 线保持高电平，表示不能接收数据。这里需要注意，主机可以作为发送器，也可以作为接收器。与此相对应，从机可以作为接收器，也可以作为发送器。

（3）终止传输。

SCL 线是高电平时，SDA 线由低电平向高电平切换，这个情况表示停止条件。在停止条件的某段时间后总线被认为再次处于空闲状态。

传输由起始条件开始，直到停止条件发生。如果传输产生重复起始条件而不产生停止条件，总线会一直处于忙的状态，此时的起始条件（S）和重复起始条件（Sr）在功能上是一样的。

主机作为发送器，可以根据情况在某一字节传输完成（从机接收器响应或不响应）后，直接产生停止条件或重复起始条件。如果主机作为接收器，接收最后一个字节后，主机接收器必须不产生响应，通知从机发送数据结束。从机发送器必须释放 SDA 线，允许主机产生停止或重复起始条件。

2. SPI 概述

SPI（Serial Peripheral Interface，串行外设接口）最早由 Motorola 公司提出，出现在其 MC68HCXX 系列单片机中。由于其简单实用，又不牵涉到专利问题，因此许多厂家的设备都支持该接口，广泛应用于外设控制领域，诸如 EEPROM、FLASH、实时时钟、A/D 转换器，还有数字信号处理器和数字信号解码器。

SPI 接口是一种事实标准，并没有标准协议，大部分厂商都是参照 Motorola 公司的 SPI 接口定义来设计的。但正因为没有确切的版本协议，不同厂商产品的 SPI 接口在技术上存在一定的差别，容易引起歧义，有的甚至无法直接互联（需要软件进行必要的修改）。

SPI 接口为全双工同步串行接口，支持单主机多从机架构。SPI 接口共有 4 组信号线，分别是设备选择线、时钟线、串行输出数据线、串行输入数据线，如下所述。

- 设备选择线/SS（或/CS）：/SS 线用于选择激活某从机设备，低电平有效，由主机驱动输出。只有当/SS 信号线为低电平时，对应从机设备的 SPI 接口才处于工作状态。
- 同步时钟信号线 SCLK：SCLK 线用来同步主从设备的数据传输，由主机驱动输出，从机设备由 SCLK 线同步接收或发送数据。
- 串行数据线（串行输出数据线、串行输入数据线）：SPI 接口数据线是单向的，共有两组数据

线，分别承担主机到从机、从机到主机的数据传输。传输数据由 SCLK 线同步进行，串行数据线可以是单线的，也可以是多线的（通常为 4 线），增加数据位宽可以大幅提升传输速率。

需要注意的是，SCLK 信号线只由主设备控制，从设备不能控制该信号线。同样，在一个基于 SPI 的设备中，至少需要有一个主控设备。这样的传输方式有一个优点，与普通的串行通信不同，普通的串行通信一次连续传送至少 8 位数据，而 SPI 允许数据一位一位地传送，甚至允许暂停，因为 SCLK 时钟线由主控设备控制，当没有时钟跳变时，从设备不采集或传送数据。也就是说，主设备通过对 SCLK 时钟线的控制可以完成对通信的控制。

不同厂商的数据线命名有所差异。Motorola 公司的经典命名是 MOSI（Master Output/Slave Input，主机输出/从机输入）和 MISO（Master Input/Slave Output，主机输入/从机输出）。这是站在信号线的角度来命名的，MOSI 线上的数据一定是主机流向从机的。因此在电路板上，主机的 MOSI 引脚应与从机的 MOSI 引脚连接在一起，双方的 MISO 也应该连在一起，而不是一方的 MOSI 连接另一方的 MISO。

不过，也有一些厂商按照类似 SDI（Serial Data Input，串行数据输入）/SDO（Serial Data Output，串行数据输出）的方式来命名，这是站在器件的角度根据数据流向来定义的。这种情况下，当主机与从机连接时，就应该用一方的 SDO 连接另一方的 SDI。

由于 SPI 接口数据线是单向的，故电路设计时，数据线连接一定要正确，必然是一方的输出连接另一方的输入。这个问题很简单，但由于不同厂商产品的命名习惯可能不同，因此需要小心，避免出现低级错误。

SPI 接口的数据传输速度总体来说要比 I2C 接口快得多，单线数据线的传输速率可达 100Mbps，4 线数据线的传输速度可达 400Mbps。

SPI 总线接口支持多从机应用。多个从机共享时钟线、数据线，可以直接并接在一起；而各从机的设备选择线则单独与主机连接，受主机控制。在一段时间内，主机只能通过某根设备选择线激活一个从机，进行数据传输，而此时其他从机的时钟线和数据线端口则都应保持高阻状态，以免影响当前数据传输的进行，如图 6-5 所示。

与 I2C 总线接口相比，SPI 协议需通过设备选择线选择设备，当出现多从机应用时，需要多根设备选择线，实现起来较 I2C 接口要复杂。此外，SPI 总线不支持总线控制权仲裁，只能用在单主机的场合，而 I2C 总线可以支持多主机应用。

SPI 接口协议相对 I2C 接口协议要简单，没有握手机制，数据传输效率高，速率也更快；此外 SPI 接口是全双工通信，可同时发送和接收数据。因此，SPI 接口比较适合用于大量数据传输的场合，比如 MMC/SD 卡的数据传输就支持 SPI 模式。而 I2C 接口协议功能较丰富，但也相对复杂，多用在传输控制命令字等有意义数据的场合。

SPI 接口属于一种非常基本的外设接口，但是应用却很广泛。SPI 也有所发展，比如 National Semiconductor 公司推出的 SPI 精简接口 Microwire，满足通常外设的扩展需求。Motorola 公司还推出了扩展功能的 QSPI（Queued SPI，队列化串行外设接口），其应用更为广泛。

3. SEP4020 中的 SPI 控制器

SEP4020 处理器的串行外设接口（SPI）模块允许在微控制器与外设间进行全双工、同步、串行的数据通信并进行数据的串并转换。SPI 模块连接在 APB 总线上，是符合 AMBA 规范的从机模块。

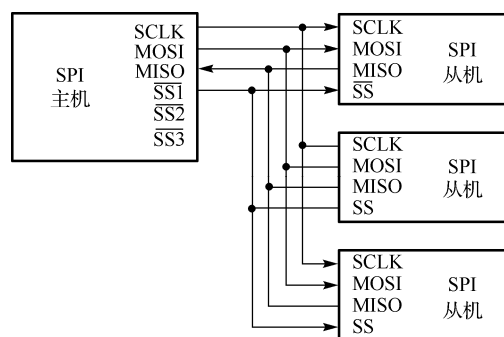


图 6-5 SPI 主机和多个从机的连接

SPI 控制器模块具有以下特征：

- (1) 符合 AMBA 2.0 接口标准。
- (2) 支持串行 Master 操作模式。
- (3) 中断可独立屏蔽，中断包括：发送 FIFO 溢出信号，发送 FIFO 空信号，接收 FIFO 满信号，以及接收 FIFO 的溢出信号。
- (4) 串行接口协议：支持 SPI（Motorola Serial Peripheral Interfacex）四线全双工串行接口协议。时钟相位、极性有 4 种组合方式，时钟相位、极性的选择决定了传输是否以第一个发送时钟作为开始，停止时钟是否保持为高电平等问题。支持 National Semiconductor Microwire 半双工的串口协议，采用控制字串行传输，来协调主设备与从设备的控制信息。

SPI 控制器模块使得 CPU 可以全双工、同步地和外部设备进行数据通信。SEP4020 中的 SPI 控制器支持 Master 工作模式，其可以发起和外设之间的一次传输。使能 SPI 控制器，SPI 输出 OUT_SCK 时钟，写 SPI 的发送 FIFO 开始一次传输。数据在时钟 OUT_SCK 的控制下通过 OUT_MOSI 串行输出，同时 SPI 在内部时钟的控制下锁存 IN_MISO 管脚上的数据并写入接收 FIFO。当发送 FIFO 中所有数据发送完毕后，SPI 中寄存器的标志位置位，SPI 使能比特位清零，输出时钟停止。

SPI 有两种工作方式：中断/软件轮询工作方式和 DMA 工作方式。

中断/软件轮询工作方式：SPI 有两个可控中断源——接收 FIFO 溢出和发送 FIFO 空（发送 FIFO 中无数据）。通过设置相应的中断使能位可以向中断控制器发送中断信号。当然，也可以禁止中断通过软件轮询来查询中断标志位。使用中断方式需要配置中断控制器。

DMA 工作方式：分为发送 DMA 请求和接收 DMA 请求。当发送 FIFO 为半空（4 个数据）时，SPI 向 DMA 控制器发送 DMA 请求直到 FIFO 中的数据个数大于 4。接收 DMA 请求又分为两种：Burst Request 和 Single Request。当接收 FIFO 为半满时，SPI 发送 Burst Request；当接收 FIFO 不空且非半满时，则 SPI 发送 Single Request。使用 DMA 方式需要配置 DMA 控制器。是否响应 SPI 的 DMA 请求以及响应哪种请求取决于 DMA 控制器的配置。

4. SPI 驱动

SEP4020 处理器中的 SPI 控制器中提供给用户配置的寄存器包括：

- 控制寄存器 0：控制帧大小、传输模式选择、串行时钟极性、串行时钟相位、帧格式、数据帧长度等。
- 控制寄存器 1：数据帧数量。
- 使能寄存器：使能 SPI。
- Microwire 控制寄存器：Microwire 握手、控制、传输模式。
- 从设备使能寄存器。
- 波特率选择寄存器。
- 发送 FIFO 阈值寄存器。
- 接收 FIFO 阈值寄存器。
- 发送 FIFO 状态寄存器。
- 接收 FIFO 状态寄存器。
- SSI 状态寄存器。
- 中断屏蔽寄存器。
- 中断最终状态寄存器。
- 中断原始状态寄存器。

- 发送 FIFO 溢出中断清除寄存器。
- 接收 FIFO 溢出中断清除寄存器。
- 中断清除寄存器。
- DMA 控制寄存器。
- DMA 发送数据寄存器。
- DMA 接收数据寄存器。
- 数据寄存器。

SPI 的驱动请见 6.3.3 节触摸屏接口小节中 SPI 与触摸屏的协同工作举例。

6.2 高速通信接口

6.2.1 通用串行总线 USB

1. USB 概述

通用串行总线（USB，Universal Serial Bus）最初是由英特尔与微软公司倡导发起的，其最大的特点是支持热插拔和即插即用。当设备插入时，主机可以侦测设备并加载所需的驱动程序。USB 规范第一次于 1995 年由 USBIF（USB Implement Forum，USB 实现论坛）提出，数据传输率为 12Mbps，支持视频数据、音频数据及普通数据的实时传输。

USB 版本经历了多年的发展，到现在已经发展为 3.0 版本，成为目前电脑中的标准扩展接口。各 USB 版本的最大传输速率各不相同，但能很好地兼容。采用旧版本 USB 的设备可以插入新版本的 USB 系统中，以旧版本的最大传输速率进行数据传输。各版本最大传输速率、最大输出电流及协议推出时间如表 6-1 所示。

表 6-1 USB 协议的发展

	最大传输速率	最大输出电流	协议推出时间
USB1.0	低速（Low-Speed） 1.5Mbps（192KB/s）	500mA	1996 年 1 月
USB1.1	全速（Full-Speed） 12Mbps（1.5MB/s）	500mA	1998 年 9 月
USB2.0	高速（High-Speed） 480Mbps（60MB/s）	500mA	2000 年 4 月
USB3.0	超速（Super-Speed） 5Gbps（640MB/s）	900mA	2008 年 11 月

需要注意的是，总线的最大传输速率与实际传输带宽有所不同，总线的最大传输速率没有考虑总线传输中的协议开销。实际情况中，对于高速设备来说，约能实现 400Mb/s（50MB/s）的数据传输。

USB 系统采用级联星型拓扑结构，如图 6-6 所示。该结构由 3 个基本部分组成：主机（host）、集线器（hub）和功能设备（function）。

（1）主机包含有主机控制器和根集线器（root hub），控制 USB 总线上的数据传输。USB 系统只能有一个根集线器，它连接在主机控制器上。主机有时也集成集线器，用于扩展主机的连接端口数量。

（2）集线器是 USB 系统的特定组成，提供端口（Port）将功能设备连接到 USB 总线上，同时检

测连接在总线上的设备，并为这些设备提供电源管理，负责总线的故障检测和恢复。通过集线器，USB 系统最多可以连接 127 个功能设备。

(3) 功能设备通过端口与总线连接。集线器与功能设备可由主机通过 USB 总线提供能源，也可由自身提供能源（通过自身外部电源接口）。需要注意的是，主机通过 USB 总线提供电源存在电流限制。USB 总线可以提供 5V 电源供连接的外设使用。USB 总线提供的电流较低，即 100~500mA 之间，对于鼠标、U 盘等外设是足够的，对于需要大电流的外设来说（比如 USB 接口的移动硬盘等），需要自身供电。

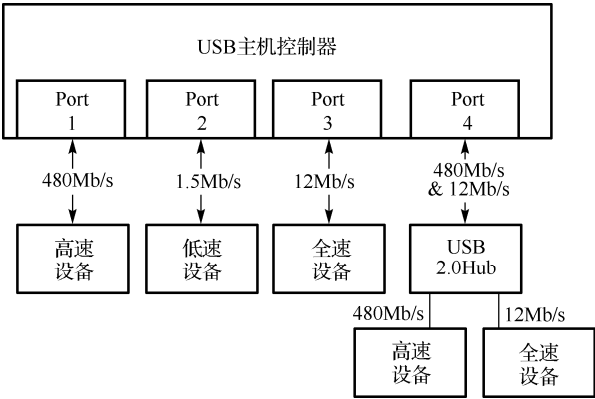


图 6-6 USB 系统的级联星型拓扑结构

USB 采用 4 针（USB3.0 标准为 9 针）接口进行连接。其中两针为电源端口（5V 及 GND），用于提供电源。另两针是使用差分信号进行数据传送的串行通道（D+ 及 D-），可以进行有效的半双工通信。传输数据通过 NRZ-I（No Return Zero-Inverse，非归零反相编码）编码，数据包传输之前使用特殊的同步序列与接收端进行时钟同步，避免了时钟线的使用。

在 NRZ-I 编码中，编码后电平只有正负电平之分，没有零电平，是不归零编码。NRZ-I 电平的一次翻转代表逻辑 0，与前一个电平相同的信号代表逻辑 1（翻转为 0，不变为 1）。根据这一编码原则，假设发送端传送 8 位数据流 0000 0001B，前面的 7 个 0 位经过 NRZ-I 编码后，将得到 7 次翻转信号，在接收端根据脉宽很容易得到同步接收时钟，此后根据这个同步接收时钟来采样后面的数据。在传输过程中，每一次编码的跳变都可以用来同步。这种同步机制在 USB 低速和全速传输中得到应用。即发送数据前，首先发送同步头，内容为 01H。这样就可以同步传输数据了，且字节开头和结尾不需要起始位和停止位。在 USB 高速传输中，同步头 SYNC 为 0001H，15 个翻转信号。当传输连续的逻辑 1 时，NRZ-I 编码后将保持上一次翻转后的状态，这使得接收端无法从中得到同步信号。为此，USB 协议规定：如果要发送的数据中出现连续的 6 个 1，则在进行 NRZ-I 编码前，在这 6 个连续的 1 后面插入 1 个 0，然后再进行 NRZ-I 编码。接收端收到连续 6 个 1，将自动去掉后面的 1 个 0，从而恢复原数据。这样就使得 USB 通信的接收同步更加可靠。

随着各种数码设备的大量普及，USB 标准得到了广泛的使用。虽然这些设备都采用 USB 接口，但是这些设备的数据线并不完全相同。这些数据线连接计算机的一端都是相同的，设备连接端通常出于体积的考虑而采用各种不同的接口。USB 是一种统一的传输规范，存在多种接口形式，最常见的就是计算机上使用的扁平的 A 型口，分为公母接口，一般线上带的是公口，机器上带的是母口。由于数码产品的体积所限，通常用的是 Mini-USB 接口或是 Micro-USB 接口，如图 6-7 所示。Micro-USB 是 USB 2.0 标准的一个便携版本，比 Mini-USB 接口更小，是 Mini-USB 的下一代接口标准。



图 6-7 Mini-USB 接口和 Micro-USB 接口

多数情况下，嵌入式系统作为 USB 功能设备使用，但在某些情况下，又需要作为 USB 主机使用。举例来说，智能手机连接计算机同步数据时，计算机作为 USB 主机，智能手机作为 USB 功能设备。当智能手机连接 U 盘等外设读取数据时，智能手机作为 USB 主机，U 盘等外设作为 USB 功能设备。USB On-The-Go (USB OTG) 技术可以很好地解决这一问题。USB OTG 模式是 USB 2.0 规范的补充，支持 USB OTG 模式的 USB 控制器可以作为主机使用，也可以作为功能设备使用。用户可以直接连接两个支持 USB OTG 模式的设备，通过连接线的定义，决定作为主机的设备。

2. USB 的数据传输

USB 规范定义了主机与设备之间的传输模式，以及传输的调度方式。USB 是基于令牌的总线，USB 主控制器广播令牌，总线设备检测令牌中的地址是否与自身分配的地址相符，通过接收或发送数据给主机来响应。端点 (End Point) 是 USB 设备或主机上的一个数据缓冲区，用来存放和发送各种数据。端点是设备的逻辑端口，一个设备可以包含 16 个端点。每一个端点都有唯一的确定地址，有不同的传输特性。

和其他数据传输协议一样，USB 数据是由二进制数字串构成的，不同的数字串排列构成域，域再构成包 (packet)，包再构成事务 (IN、OUT、SETUP)，事务最后构成传输。域是 USB 数据最小的单位，由若干位组成 (具体位数由域决定)，域可分为 7 个类型：

(1) 同步域 (SYNC)：8 位，值固定为 01H (对于高速传输为 16 位，值固定为 0001H)，用于本地时钟与输入同步。

(2) 标识域 (PID)：由 4 位标识符+4 位标识符反码构成，表明包的类型和格式，这是一个很重要的部分。

(3) 地址域 (ADDR)：7 位地址，代表了设备在主机上的地址，地址 0 被命名为零地址，是任何一个设备第一次连接到主机时，在被主机配置、枚举前的默认地址，由此可以知道为什么一个 USB 主机只能接 127 个设备的原因。

(4) 端点域 (ENDP)：4 位，由此可知一个 USB 设备具有的端点数量最大为 16 个。

(5) 帧号域 (FRAM)：11 位，每一个帧都有一个特定的帧号，帧号域最大容量 0x800，对于同步传输有重要意义 (同步传输为 USB 四种传输类型之一，将在下文介绍)。

(6) 数据域 (DATA)：长度为 0~1023 字节，在不同的传输类型中，数据域的长度各不相同，但必须为整数个字节长度。

(7) 校验域 (CRC)：对令牌包和数据包 (包的分类将在后文介绍) 中非 PID 域进行校验的一种方法。CRC 校验在通信中应用很广泛，读者可自行查阅相关资料。

由域构成的包有 4 种类型，分别是令牌包、数据包、握手包和特殊包，前面 3 种是重要的包，不同的包的域结构不同，分别介绍如下。

(1) 令牌包：可分为输入包、输出包、设置包和帧起始包 (注意这里的输入包是用于设置输入命令的，输出包是用来设置输出命令的，而不是用来放数据的)。其中输入包、输出包和设置包的格式都是一样的：SYNC+PID+ADDR+ENDP+CRC5 (5 位校验码)。帧起始包的格式为：

SYNC+PID+FRAM+CRC5（5 位校验码）。

（2）数据包：分为 DATA0 包和 DATA1 包，当 USB 发送数据的时候，当一次发送的数据长度大于相应端点的容量时，就需要把数据包分为好几个包，分批发送，DATA0 包和 DATA1 包交替发送，即如果第一个数据包是 DATA0，那第二个数据包就是 DATA1。但也有例外情况，在同步传输中（四种传输类型之一），所有的数据包都是 DATA0，格式为：SYNC+PID+0~1023 字节数据+CRC16（16 位校验码）。

（3）握手包：结构最为简单的包，格式为：SYNC+PID。

USB 数据传输过程如图 6-8 所示。

（1）主机控制器首先产生令牌包（token packet），表明传输类型、传输方向（传输方向由主机的端点决定，输出传输表明数据由主机发往设备，输入传输表明数据由设备发往主机），以及目标设备的端点地址。

（2）目标设备接收令牌包后，根据令牌包指定的传输方向，返回数据或是接收来自主机的数据。

（3）数据传输完成后，握手包用于表明传输完成情况。对于输出传输，响应信号由设备产生；对于输入传输，响应信号由主机产生。

主机与设备端点之间的逻辑通道称为管道（pipe），主机与设备之间的数据传输通过管道进行。管道可以是单向的，也可以是双向的。管道表示主机运行软件与相应设备端点之间逻辑关联，通过管道的抽象，软件可以忽略 USB 系统层次结构，直接访问设备端点。

USB 传输协议定义了 4 种基本传输类型，4 种传输类型具有不同的性质，使得 USB 传输可以在多种设备中得到多样性的使用。4 种基本传输类型定义如下。

（1）控制（control）传输：支持外设与主机之间控制、状态、配置等信息的传输，为外设与主机之间提供一个控制通道。每种外设都支持控制传输类型。控制传输常用于建立其他外设端点。

（2）块数据（bulk data）传输：支持 U 盘、打印机、数码相机等外设，这些外设与主机间传输的数据量大，但是没有严格的传输延时限制。块数据传输所占的带宽随其他设备对于总线的使用情况而变化。块数据传输的优先级最低。

（3）中断数据（interrupt data）传输：支持游戏手柄、鼠标和键盘等输入设备，这些设备与主机间数据传输量小，无周期性，但对响应时间敏感，要求马上响应。中断数据传输还可以请求块数据传输，无需等待主机轮询端点。

（4）等时（isochronous）传输：也称为同步传输。支持周期性的、延时受限且带宽固定的数据传输。等时传输分配有相应的总线带宽，通常用于音、视频等实时数据传输。

管道只能配置进行一种传输类型，无法同时进行多种类型混合传输。主机控制器在管道间，根据不同的传输类型分配带宽。进行等时传输或中断数据传输的端点分配有保留带宽，并保证数据按一定的速率传送。进行控制传输或块数据传输的端点按可用的最大带宽来传输数据。

3. USB 主机与设备之间的交互

主机控制器启动时，或是设备插入后，主机控制器将开始总线枚举（bus enumeration）过程。USB 支持热拔插功能，连接设备或移除设备时无需考虑供电状态。插入设备后，主机可以检测出总线上电

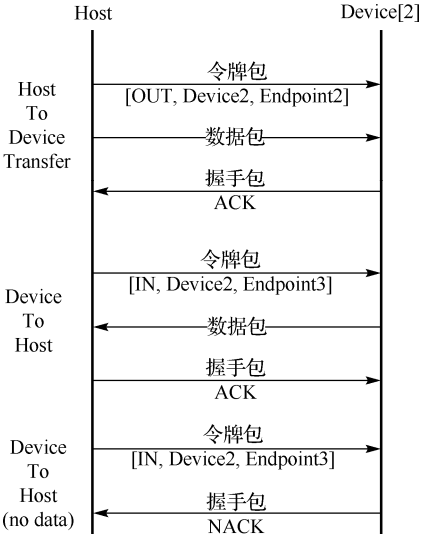


图 6-8 USB 数据传输过程

气特性的改变，触发设备的安装过程。主机可以实时检测插入总线的设备，开始总线枚举。在总线枚举的过程中，主机将为插入的设备分配地址，加载相应的驱动。USB 传输协议内建错误处理及故障恢复机制，具有较强的健壮性。

上文提到数据是在管道中传输的，管道连接主机与设备端点。设备至少带有一个端点，即 0 号端点。0 号端点可以进行控制传输。通过该端点，主机可以与设备实现交互。进行控制传输的端点需要支持以下命令：`set_address`（地址设置）、`set_configuration`（参数配置）以及 `get_descriptor`（获取描述符）。描述符即设备的标准信息，所有 USB 设备均带有如下信息：

- 厂商标识。
- 设备类型。
- 电源管理能力。
- 端点描述及配置信息。

除此之外，根据设备类型的不同，还存在其他标准信息。另外，厂商也可以增加关于设备的额外信息，这些信息由厂商自行决定，没有标准规范。

根据不同的端点使用情况，设备存在不同的配置选择。作为最简单的 USB 设备，U 盘只需使用两个端点进行块数据传输（一个输入，一个输出），U 盘只有一种配置。与 U 盘不同，智能手机与计算机相连时，可以进行多种不同类型的数据传输，诸如通过图片传输协议（Picture Transfer Protocol）进行静态图像捕捉，进行音频流数据的处理，或进行简单的块数据传输。因此智能手机多存在多种配置，不同配置使用不同的端点，一个时刻只能激活一种配置。

对于特定配置的设备来说，设备描述符将指明使用的接口类（`interface class`）以及接口子类（`interface subclass`）。接口类及接口子类说明端点的传输方式，根据不同的通用用途分类标准化。操作系统一般带有标准设备的通用驱动，一旦设备符合相应的接口标准，主机无需安装特殊的设备驱动。如果设备不属于标准设备，或不符合相应的接口标准，则主机需要安装相应的设备驱动，以实现设备交互。USB 规范支持以下标准设备接口：

- 蓝牙设备。
- 智能卡接口。
- USB 集线器。
- 鼠标、键盘等输入设备。
- 大容量存储器。
- 打印机。
- 扫描仪、照相机（使用图片传输协议）。
- 摄像机（媒体传输协议）。
- USB 声卡。
- 调制解调器。

USB 规范实现各类应用的标准化，包括所需端点，端点描述符，以及传输数据的具体格式。

上文提到，各个传输管道的传输带宽由主机控制器分配。USB 传输以帧（`frame`）为单位分配带宽，每帧为 1ms。对于 USB 高速传输，每帧细化为 8 微帧，每微帧为 125μs。每帧/微帧可以包含多次 USB 传输，一次传输无法跨越多个帧/微帧。USB 数据传输以包的形式进行，包的大小存在最大限制，由端点描述符决定。这样，主机控制器在每帧/微帧内调度传输，建立传输列表，以此分配各管道带宽。

4. SEP4020 中的 USB 控制器

USB 模块作为 USB 协议中定义的设备（`USB DEVICE`）控制模块，负责与 USB 协议中定义的

制定了 DLX 标准，即 10Mbps 普通以太网标准。1983 年，IEEE 组织公布了 IEEE802.3 标准，10Mbps 以太网得到了国际的认可，标志着以太网标准的正式成立，同时也标志着符合国际标准的以太网技术的面世。

1995 年 IEEE 通过了 IEEE802.3u 标准，标志着 100Mbps 快速以太网时代的到来，随着网络用户的日益增加，10Mbps 带宽在一些环境下不能满足人们对信息传输速率的要求。因此快速以太网推出后得到了广泛的推广。快速以太网是在 10Mbps 普通以太网基础上发展起来的，它在保持帧格式、介质访问控制的前提下，工作速率比普通以太网快 10 倍。IEEE802.3u 快速以太网标准是 IEEE802.3 标准的补充，并且兼容 10Mbps 以太网。10Mbps 和 100Mbps 以太网组网方便、价格低廉、性能高效，在局域网中受得到了广泛应用。1998 年通过了 IEEE802.3z 标准，以太网的运行速度达到 1Gbps，目前已出现了 10Gbps 的以太网。

与传统的 CAN、RS-485 等相比，以太网速度更快、通用性更好，而且能直接与因特网相连，具有更大范围的远程访问能力。相对于 USB、IEEE1394 等总线，以太网在传输距离和控制软件的通用性上有明显的优势。

1. 10Mbps 和 100Mbps 以太网的体系结构

MAC（Media Access Control）为介质访问控制，对向通信介质中发送信息或者从通信介质中接收信息的行为进行控制。它有 3 个主要的功能：①决定站点何时开始发送数据帧；②将数据帧发送到 PHY 上并转换为以太网包；③接收来自 PHY 的数据，传给 MAC 客户端。

MAC 的一个重要的特点是 MAC 站点访问介质时，具有介质无关性。MAC 和网络中使用的介质无关，它通过以太网 PHY 访问介质。图 6-11 给出了 MAC 在参考模型中的位置。

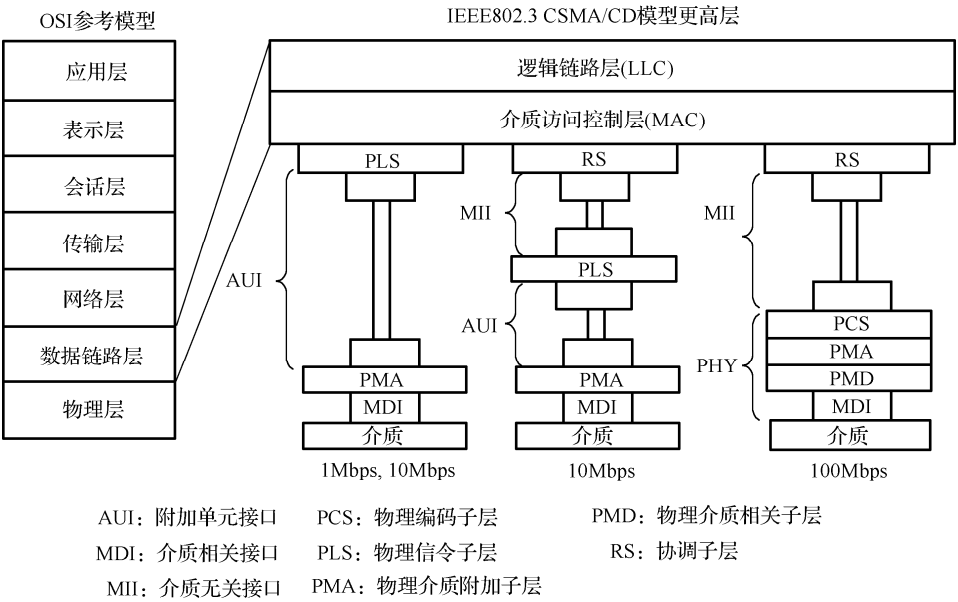


图 6-11 OSI 参考模型与 IEEE802.3 参考模型

100Mbps 以太网又称为快速以太网，是在 10Mbps 普通以太网的基础上发展起来的。快速以太网是 10Mbps 的改进，同时兼容 10Mbps 以太网。100Mbps 以太网与 10Mbps 以太网的主要区别如下：

- (1) 快速以太网协议中增加了全双工和硬件流量控制。
- (2) 10Mbps 以太网只能工作在单一速率，而快速以太网能工作在 10Mbps 和 100Mbps 两种速率上。
- (3) 100Mbps 以太网中的与介质无关接口（MII，Media Independent Interface）取代了 10Mbps 以

以太网中的附加单元接口（AUI, Attachment Unit Interface）。AUI 能减弱 MAC 对各种不同 PHY 的要求，但是在 100Mbps 的速率下频率过高，AUI 并不能完成上述功能。MII 接口在发送和接收数据时采用的是 4 位数据（半组元）接口，这就使得频率的要求从 AUI 接口时的 100MHz 降到 25MHz（10Mbps 时为 2.5MHz）。MII 信号提供了网络错误信号和对 PHY 的管理信号，能工作在 10Mbps 和 100Mbps 的速率上。

（4）快速以太网协议中增加了协调子层（RS, Reconciliation Sublayer）。它位于最初的 MAC 层和 MII 之间，MAC 只能提供位串行接口，而 MII 可以提供一个具有半位元组宽度的发送和接收数据接口，RS 提供了一个在原始以太网 MAC 和 MII 之间的映射。以太网的 MAC 应保持一个标准实体，虽然 PHY 可以改变，但 MAC 是以太网的核心，RS 相当于连接器将最初的位串行 MAC 接口转换为 MII 接口。

（5）快速以太网增加了自动协商（也称自适应）功能。同时支持 10Mbps 和 100Mbps 工作速度的以太网 MAC、RS 和 MII 之间在功能上有很程度的共享性，自动协商不需要用户的干预，是一种可以在进行网络连接之前自动探测出网络设备正常工作速度的机制。

（6）快速以太网中采用了简单不归零码（NRZ）。AUI 中曼彻斯特编码会在高频下带来一些负面效应，特别是在数据速率增加时影响更大，因为电磁干扰和射频干扰在高频下会很严重。MII 的发送和接收数据通路中采用简单不归零码。

1）以太网的帧格式

以太网上的站点发送数据都是按以太网帧的格式在总线上进行传输的，以太网帧是网络通信的基本单元。10Mbps/100Mbps 以太网 MAC 帧格式见表 6-2。

表 6-2 以太网帧格式

字节数	7	1	6	6	2	46~1500	4
内容	Pre	SFD	DA	SA	Lengh/Type	Data unit + pad	FCS

- Pre (Preamble)：前导码，7 字节。Pre 字段中 1 和 0 交互使用，接收站通过该字段知道导入帧，并且该字段提供了同步化接收物理层接收部分和导入比特流的方法。
- SFD (Start-of-Frame Delimiter)：定界符，1 字节。字段中 1 和 0 交互使用，结尾是两个连续的 1，表示下一位是利用目的地址的重复使用字节的重复使用位。
- DA (Destination Address)：目的地址，6 字节。DA 字段用于识别需要接收帧的站。
- SA (Source Address)：源地址，6 字节。SA 字段用于识别发送帧的站。
- Length/Type：长度/类型，2 字节。如果采用可选格式组成帧结构，该字段既表示包含在帧数据字段中的 MAC 客户机数据大小，也表示帧类型 ID。
- Data unit+pad：数据和填充域，是一组 n ($46 \leq n \leq 1500$) 字节的任意值序列，帧总值最小为 64 字节。
- FCS (Frame Check Sequence)：帧校验，4 字节。该序列包括 32 位的循环冗余校验（CRC）值，由发送 MAC 方生成，通过接收 MAC 方进行计算得出以校验被破坏的帧。

2）半双工以太网

IEEE802.3 规定 MAC 对介质访问控制采用的规则是带冲突检测的载波监听多路访问协议 CSMA/CD（Carrier Sense Multiple Access With Collision Detection）。CSMA/CD 是一种介质访问控制协议，包括 3 方面的含义：

- 载波侦听。网络上的站点要能连续监听网络，载波信号有效时表示网络忙，否则网络处于寂静时期。

- 多路访问。网络上的站点检测到网络空闲时，都可以向网络中发送数据包。
- 冲突检测。当两个以上的站点在一段寂静时间后开始发送时，则会产生冲突，网络上的每个站点都能检测到传输介质中的冲突。

帧的发送过程：帧发送时，首先进行数据帧的组合，将前导码、定界符、目的地址、源地址、数据和填充码、帧校验按照以太网帧格式进行封装，按照 CSMA/CD 协议进行发送。

帧的接收过程：当接收数据帧时，MAC 检查前导码和帧的定界符，当接收到定界符 SFD 时表示帧接收的开始。首先进行的是目的地址的判别，若接收的地址不是站点的地址或者广播地址则放弃本次接收。如果地址相符，则进行数据帧长度的判断，如果帧长度小于最短帧则放弃本次接收；如果帧的长度大于最长帧，将会产生帧过长错误。在完成数据的接收后，接收站点进行 CRC 校验计算，和接收帧中的 FCS 校验码进行比较，如果校验错误，MAC 会产生一个错误，如果字节数正确，则产生 CRC 错误；如果接收的位数不正确，则产生对齐错误。如果帧校验正确，则对数据帧进行解封，去除发送时的附加位，将数据提交给上层。

MAC 正常接收帧的时序如图 6-12 所示。

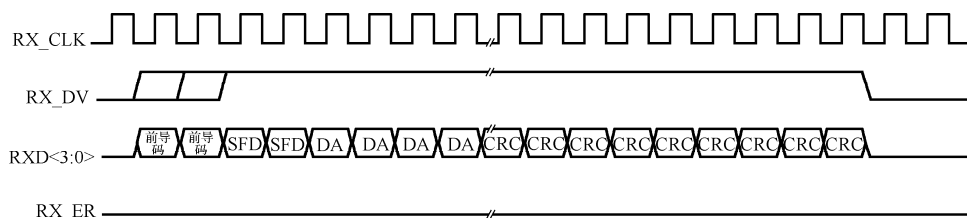


图 6-12 正常接收帧

2) 全双工以太网

全双工模式下允许发送站点和接收站点采用点对点的连接，介质提供独立的发送数据和接收数据路径。全双工模式下的站点之间不受对方站点发送数据的干扰而独立工作。在该模式下发送数据帧时，将忽略载波侦听，没有传输延时，忽略冲突信号。CSMA/CD 算法在全双工模式下帧发送时，帧与帧之间只要等待一个帧间隙时间，而在接收时和半双工帧的接收过程是一样的。工作在全双工模式下需要满足以下条件：

- 介质需支持无冲突的发送和接收。
- 仅有两个站点以点对点的方式连接，无介质的竞争。
- 两个设备工作在全双工模式。

以太网帧无错传输的概率很高，但也无法保证帧传输的完全正确性。在数据出错、接收缓冲区溢出或者其他异常情况下，以太网接收会简单地丢弃帧而不会有任何提示，缓冲拥塞造成的丢帧在高速率情况下的概率是很大的。在全双工以太网中由于没有冲突检测，载波侦听和退避算法都不能工作，当网络中的流量超过网络的传输能力时，网络的吞吐量就会下降，所以全双工需要流量控制机制。IEEE802.3x 是专门用于全双工以太网流量控制算法的规范，也称为 Pause 控制规范。IEEE802.3x 没有定义一个全双工以太网的流量控制协议，而是规定了一个控制以太网 MAC（MAC 控制结构）的体系架构。

MAC 控制是数据链路层的一个子层，介于传统以太网 MAC 层和 MAC 客户之间。客户是网络层协议或者数据链路层里实现转发功能的网桥中继实体和 LLC 等。图 6-13 给出了 MAC 控制层在体系结构中的位置。

当 MAC 客户发出流量控制请求后，MAC 将产生 MAC 控制帧（或称 Pause 帧），控制帧将发送到底层的以太网上。在接收方，以太网 MAC 将接收 MAC 控制帧，然后交给 MAC 控制子层。此时 MAC 执行暂停发送以太网帧操作，当暂停时间到达控制帧中设定的暂停值后继续发送以太网帧。

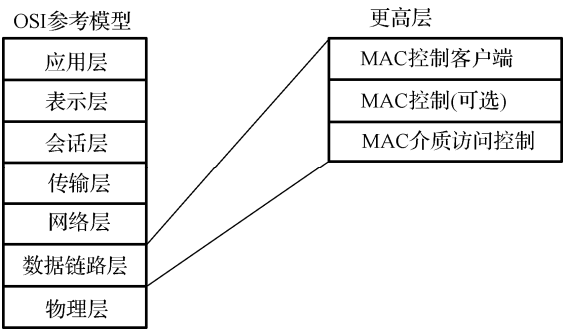


图 6-13 MAC 控制在参考模型中的位置

4) 以太网间隙时间

以太网的间隙时间包括两层含义。第一层含义是相距最远的站点间的信号来回所需要的时间，称为物理循环传播时间。第二层含义为检测到冲突并发出冲突的强制阻塞序列所需要的时间。将这两层含义的时间相加，再加入一些附加时间进行处理得到一个传输 512 位的时间，也称为时间槽。

间隙时间主要有以下几方面的作用：

(1) 间隙时间为站点获得共享以太网信道建立了最大上限。站点发送数据帧的时间长度能够使最大范围内的站点都能够侦听到，并且在冲突时能够使得冲突信息从最远端返回站点，则说明此站点获得了信道，在帧的剩余部分传输过程中不会产生冲突，这时，其他的站点都侦听到了载波信号。间隙时间将获取共享信道的上限定为 512 位时间。

(2) 合法的冲突只能在间隙时间里。网络中所有的站点监测到信道中的载波会放弃帧的发送。所以这对冲突帧的碎片提供了上限（即小于 512 位），站点检测到冲突产生的碎片时将其丢弃。

(3) 帧间隙时间还将有效 MAC 帧的最小长度定义为 64 字节，其中包括 6 字节的地址，6 字节的源地址，2 字节的类型/长度，46 字节的数据，以及 4 字节的 FCS。

(4) 间隙时间是执行退避算法的基本时间单位。

2. SEP4020 中的 10/100M 以太网（RMII MAC）控制器

SEP4020 提供 MAC 的 RMII 接口，MAC 控制器支持全双工、半双工模式下的 10Mbps 和 100Mbps 自适应传输，同时支持独立多址、多地址和广播地址的地址模式，可配置实现数据帧和控制帧的接收和响应。其功能描述如下：

- 兼容 IEEE 802.3 和 802.3u 标准，支持 10/100M 自适应以太网。
- 仅支持 RMII 接口。
- 自动 CRC 填充和校验。
- 自动抛弃错误帧。
- 支持地址过滤和网络监听。
- 支持全双工、半双工模式，支持硬件流控。
- 支持超短帧丢弃，支持超长帧自动截取。
- 兼容 AMBA 2.0 协议的 AHB MASTER 接口和 AHB SLAVE 接口。
- 发送接收独立的 32×8 FIFO。
- 支持芯片外接 PHY 的管理。

MAC 控制器支持全双工、半双工模式下的 10Mbps 和 100Mbps 自适应传输，同时支持独立地址、多地址和广播地址的地址模式，可配置实现数据帧和控制帧的接收和响应。

全双工模式：全双工操作允许一对节点设备使用点对点的连接实现同时传输。因为没有对传输介质的争夺，因而相对于半双工模式没有延迟传输，没有载波检测，没有接收反馈。全双工模式主要用于网络设备中连接桥端口和独立网络设备的中心桥（即交换机）。全双工操作需要满足以下条件：物理介质支持没有冲突的同时发送和接收；有且仅有两个设备以点对点的方式连接。没有对介质的竞争，因而不需要多接入（CSMA/CD）算法；两个设备能够并且确认在使用全双工的操作。

半双工模式：半双工模式中，数据是通过在共享介质上采用载波监听多路访问/冲突检测（CSMA/CD）协议实现传输的。两个以上的设备共用传输介质，采用分布式控制方法，附接介质的各个设备通过竞争的方式获得介质的使用权。只有获得使用权的设备才可以向介质发送信息帧，该信息帧将被附接介质的所有设备感知。当发送时，设备等待（延时）一段时间直到没有其他设备发送，然后以比特流的形式发送信息。如果在开始发送时与另一个设备发生冲突，那么每个设备都会故意发送额外附加的周期保证冲突在系统内传播。在重新发送之前设备会等待一段随机时间。它的主要缺点在于有效性和距离限制，链路距离受最小 MAC 帧大小的限制，该限制极大地降低了其高速传输的有效性。

载波侦听（半双工）：发送设备在发送信息帧之前，必须侦听媒体是否处于空闲状态。如果有信号传输，则继续监听；如果没有信号传输，若则发送数据的同时继续对媒体监听。如果发送的数据与监听的数据不一致，则停止数据传输，并报告冲突。如果发送的数据与接收的数据一致，则认为数据发送正确，没有冲突发生，发送任务完成。

多路访问（半双工）：多路访问具有两种含义，即表示多个结点可以同时访问媒体，也表示一个结点发送的信息帧可以被多个结点所接收。设备发送的信息帧可以为独立地址、组地址和广播地址，接收设备在收到信息帧后根据地址接收模式（独立、组合和广播地址）接收或放弃该帧。

碰撞检测（半双工）：发送结点在发出信息帧的同时，还必须监听媒体，判断是否发生冲突（同一时刻，有无其他结点也在发送信息帧）。若碰撞窗口内发生了碰撞，则停止数据发送，同时发送额外附加的周期，以使其他设备也感知冲突，在等待一段时间后重新准备发送。如果在碰撞窗口后发生碰撞，就放弃当前的传输帧并报告为“延迟碰撞”。

数据发送：当系统有数据需要通过以太网发送时，首先通过 AHB 总线将发送数据传送给 MAC，MAC 将数据按照以太网数据帧的格式打包，主要是加上 PREAMBLE、SFD 和 FCS 校验，然后按照 IEEE 802.3 协议以 Nibble（4 比特，半字节）的格式将数据发送给 PHY，PHY 在收到数据后转换成模拟差分信号送到以太网。

数据接收：当 MAC 收到从 PHY 来的数据后，先检查帧的 Preamble、SFD 和目的地址，通过判断 Preamble 和 SFD 检查接收帧是否是有效的 MAC 帧，如果不是则抛弃该帧，之后对目的地址进行判断。接收设备可以通过配置实现对各种地址（广播地址、组地址和独立地址）的不同响应。比较接收地址与本机地址的配置，如果一致，将数据接收到 FIFO，如果错误，则将帧丢弃。最后检查接收数据的 CRC 是否正确。

帧描述符：帧描述符（Buffer Descriptors）分为发送帧描述符和接收帧描述符。SEP4020 的 MAC 控制器帧描述符其实是控制器内部的一块 RAM 存储器，用于保存带发送数据的描述信息以及接收到数据的描述信息。每个描述符占用 64 位（8 字节），共 128 个描述符，共占用 1024 字节的存储空间，程序员可以通过相应的地址读取（通常用于接收）和写入（通常用于发送）描述符。发送帧描述符用于对发送数据进行配置并指示出发送过程的状态。描述符的高四字节用于存放等待发送的数据在内存缓冲区中的地址，第五字节和第六字节指示发送数据的长度。当第七和第八个字节用于描述发送控制信息，其中第 15 位置 1 时，表示此帧描述符已经配置好可以用于发送。接收帧描述符用于对接收数据的检查并指示出接收过程的状态。描述符的高四字节指出接收数据在内存缓冲区中的存放地址，第五

和第六字节指示接收到的数据长度。第七和第八个字节用于存放接收到数据的相关控制信息，其中第 15 位置 1 时，表示此帧描述符已经配置好可以用于接收。

PHY 管理：SEP4020 通过简单的串行单数据接口，实现了 MAC 对 PHY 的配置和状态读取。本芯片的 MAC 控制器可以实现对 PHY 的 3 项命令支持。写控制字：将控制字写到 PHY 的配置寄存器；读状态：读 PHY 的控制和状态寄存器；扫描状态：持续读取 PHY 的状态寄存器。

中断：MAC 控制器提供一个中断请求信号给 SEP4020 的中断控制器，该中断可屏蔽，通过中断源寄存器可以查询中断的类型，中断源有：发送/接收完成中断、发送/接收出错中断、接收数据溢出中断。

以太网 MAC 控制器的总体结构如图 6-14 所示。

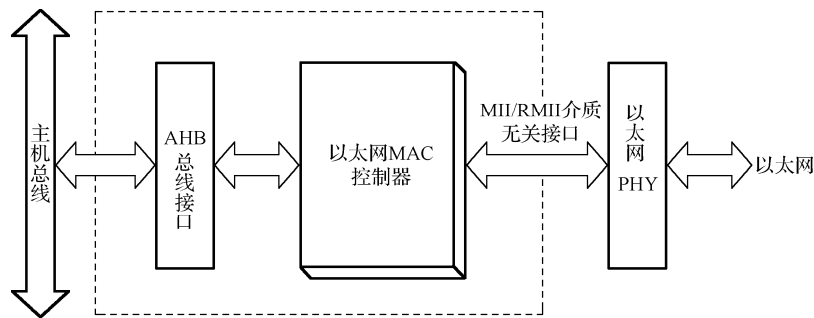


图 6-14 以太网 MAC 控制器的整体框图

IEEE802.3 以太网 MAC 控制器的设计是在 IEEE802.3 以太网协议的基础上设计一个 MAC 层的以太网控制器，能够在 10Mbps 和 100Mbps 速率下按照 CSMA/CD 协议发送数据帧和接收数据帧。发送时，通过总线接口从发送存储区中取出相应的数据到 AHB 总线接口中的发送 FIFO 中，在发送模块封装以太网帧，根据协议中的发送机制，在相应的发送时钟下通过 MII 接口或者 RMII 接口发送到物理层 PHY 中去，同时能够对 PHY 进行读写管理。在接收时，能从 PHY 中接收帧，按照协议中的接收机制在接收模块对帧进行识别和解封，将帧中的有效数据通过总线接口中的接收 FIFO 提交到上层主机系统。在发送和接收时能够支持全双工和半双工的工作模式，同时能够进行内部回环（LOOPBACK）工作。根据对 MAC 功能的定义，将整个 MAC 控制器表示为图 6-15 所示的结构框图，主要模块有：发送模块，接收模块，MAC 控制模块，MII 管理模块，寄存器模块，状态模块，RAM 模块，总线接口模块，等等。

- （1）发送模块：主要是将上层协议提供的数据封装之后发送到以太网 PHY 中。通过总线接口从存储器中读取要发送的数据，将帧封装后，在 PHY 提供的载波侦听和冲突检测信号空闲时，将数据发送到网络上。
- （2）接收模块：完成 MAC 的接收功能。接收外部以太网 PHY 的数据帧，完成对以太网单播地址、多播地址和广播地址的过滤，帧碎片滤除，将有效的数据提交给上层。
- （3）MAC 控制模块：主要负责流量控制，用于在接收缓冲区溢出时发送控制帧或接收控制帧进行暂停发送操作。
- （4）MII 管理模块：MII 管理主要负责配置和读取外部以太网 PHY 的状态。
- （5）寄存器模块：主要负责配置 MAC 的工作状态以及对 PHY 的读写操作状态。
- （6）状态模块：保存发送和接收产生的状态。
- （7）总线接口模块：MAC 通过 AHB 总线和主处理器之间的通信，在主机上可以实现更高层次的以太网协议。
- （8）描述符 RAM：存储接收和发送描述符。

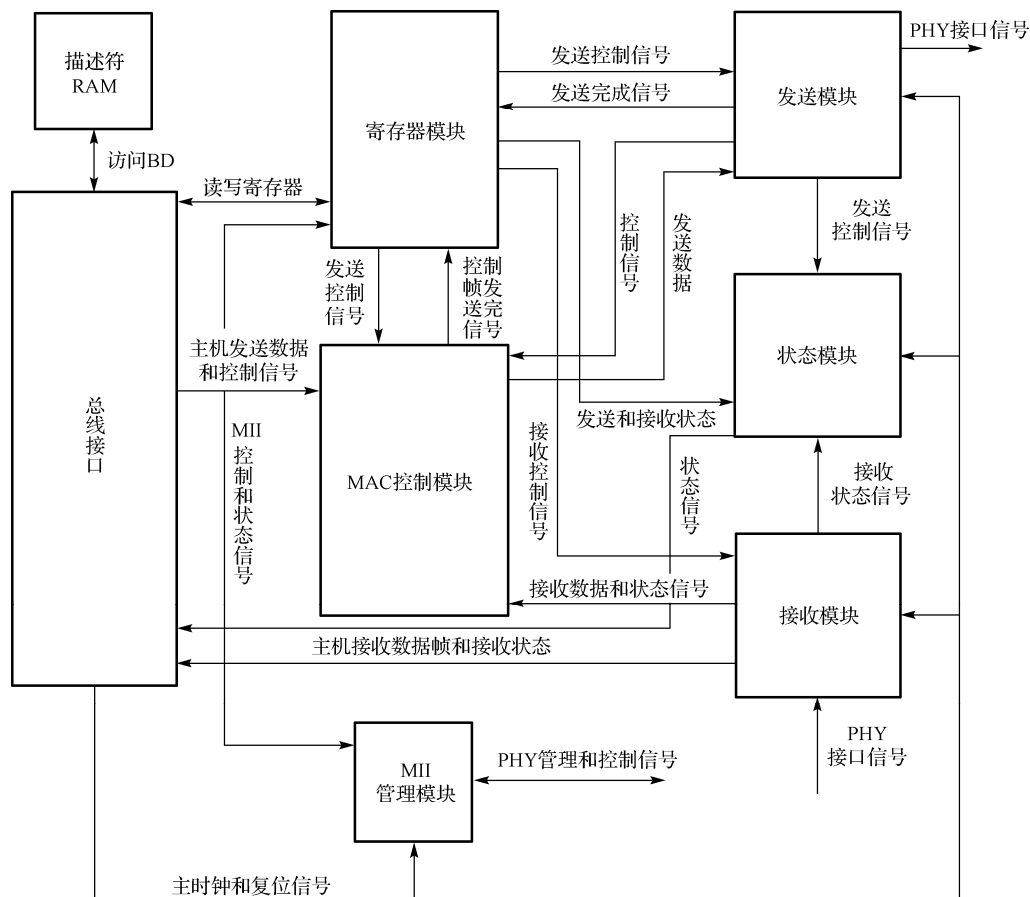


图 6-15 以太网 MAC 控制器的结构

3. SEP4020 处理器的 MAC 驱动

SEP4020 处理器中的 MAC 控制器中有如下寄存器。

- (1) 控制寄存器：设置 MAC 控制器的工作参数。
- (2) 中断源寄存器：当中断产生后，可以通过读取该寄存器值判断产生中断的原因，通过写 1 将寄存器相应位清零。
- (3) 中断屏蔽寄存器：可以对相应的中断源进行屏蔽。
- (4) 连续帧间隔寄存器：连续帧间隔是指连续发送两个数据帧之间空闲的时间长度，802.3 协议推荐为 100M 0.96 μ s、10M 9.6 μ s。全双工模式下，配置值为发送时钟周期数减去 3；半双工模式下，配置值为发送时钟周期数减去 6。
- (5) 等待窗口寄存器 1：如果在该窗口时间内出现载波，那么发送等待需要重新开始，如果在该窗口时间之后出现，则等待计数器继续计数。
- (6) 等待窗口寄存器 2：在 100M 下等于 0.96 μ s，在 10M 下等于 9.6 μ s。
- (7) 帧长度寄存器：可根据 802.3 协议配置最小帧长度和最大帧长度。
- (8) 碰撞重发寄存器：配置在发送时如果遇到数据碰撞时的最大重发次数。
- (9) 发送描述符数目寄存器：可配置发送描述符的个数。
- (10) 流控寄存器：可配置是否允许发送流控帧和是否允许接收流控帧。

(11) PHY 控制寄存器：配置发送管理帧到 PHY 时，是否需要添加 PREAMBLE，以及管理帧发送时钟分频数。

(12) PHY 命令寄存器：发送命令给 PHY，写 PHY 寄存器、读 PHY 寄存器、扫描 PHY 寄存器。

(13) PHY 地址寄存器：配置 PHY 内部的寄存器地址。

(14) PHY 写数据寄存器：PHY 寄存器写操作时写入的值。

(15) PHY 读数据寄存器：PHY 寄存器读操作时读出的值。

(16) PHY 状态寄存器：读该寄存器可读取到 PHY 的状态，如是否可以进行新的 PHY 寄存器读写操作等。

(17) MAC 地址寄存器 0：MAC 地址的第 3~6 个字节。

(18) MAC 地址寄存器 1：MAC 地址的第 1~2 个字节。

(19) MAC HASH0 寄存器：HASH 表用于进行组播的支持，通过 HASH 算法可以用少的寄存器实现大范围地址的转换。可以将多位地址的比较变成少量数据的匹配问题。SEP4020 芯片为所有的地址选择了统一的 HASH 算法。该寄存器用于配置 HASH0 的值。

(20) MAC HASH1 寄存器：该寄存器用于配置 HASH1 的值。

(21) MAC 控制帧寄存器：对寄存器的某位写 1 请求发送控制帧，控制帧发送后该位自动清零；配置控制帧中的时间参数，即请求对方暂停发送数据的时间~

(22) 发送描述符寄存器：配置等待发送数据存放的地址、发送数据的字节长度、数据发送后是否发出中断、当前描述符是否是描述表中的最后一个描述符、如果数据帧长度没有达到最小长度是否填充、是否添加 CRC 校验在数据帧尾等。

(23) 接收描述符寄存器：配置接收数据的存放地址、数据的字节长度、数据接收后是否发出中断、当前描述符是否是描述符表中的最后一个、接收到的数据帧是否和 MAC 地址一致、是否有 CRC 错误、是否在碰撞窗口后发生碰撞等。

SEP4020 的 MAC 是可以适应 10M 和 100M 两种工作模式的，但由于采用 RMII 的 PHY 接口，MAC 的工作主频是直接由片外的 50MHz 直接提供的，因此正确的软件流程应该是 PHY 芯片自适应网络的情况，由 PHY 将自己的工作频率选定为 10M 或者 100M，全双工或半双工，然后软件应该通过读取 PHY 芯片中的一个状态寄存器来获知网络的实际工作速度，并根据这个速度对 MAC 的控制寄存器进行配置，使 MAC 可以获得正确的工作频率。

初始化 MAC 需要进行如下步骤：

(1) 配置 MII 寄存器，用于配置 PHY 的工作模式，包括传输速率（10M/100M）、全/半双工模式、自适应、LED 配置。

(2) 初始化所有接收和发送描述符寄存器，最好先把所有的 BD（Buffer Descriptor，接收描述符和发送描述符）使能置 0。

(3) 配置 MAC 地址。

(4) 配置中断寄存器。

(5) 配置描述符个数的分配（系统默认为 64 个发送描述符和 64 个接收描述符）。

(6) 为接收配置相应的描述符寄存器。

(7) 配置 MAC 控制寄存器，MAC 开始工作。

(8) 当有数据要发送时，配置发送描述符，描述符的地址指针指向需要发送的数据地址。

MAC 发送过程需要进行如下编程：

(1) 从上层协议栈的发送队列中得到此次发送的长度和地址。

- (2) 搜索一个可用的发送描述符。
- (3) 设置得到的可用描述符相应位, 如果需要, 复制待发数据到描述符指定的发送缓冲区中。
- (4) 控制器硬件将按照发送描述符的相关信息发送帧数据。

MAC 接收过程需要进行如下编程:

- (1) 搜索接收数据的描述符, 从上一次的位置查询开始, 直到第一个不可用的接收描述符为止。
 - 获得接收数据长度。
 - 获得该描述符的数据存放地址。
 - 将接收指针保存下来, 后面会用到。
 - 重置相应的 `RX_BD` 用于接收数据。
 - 接收描述符索引号增加。
 - 获得 `BUFFER` 空间。
- (2) 激活接收高层的软中断。

MAC 的中断函数需要进行如下编程:

- (1) 读取 MAC 中断状态寄存器。
- (2) 写 1 清除状态位。
- (3) 如果是发送数据中断, 记录发送错误或者发送成功, 激活发送 HSR 中断。
- (4) 如果是接收数据中断, 记录接收错误或者接收成功, 开始接收数据。
- (5) 如果是接收溢出中断, 进入异常处理程序。

6.3 人机接口

6.3.1 液晶显示器接口

1. 液晶显示系统的基本原理

液晶是一种介于液体和晶体之间的物质。因为可以通过电流来改变分子结构, 所以我们可以为液晶加上不同的工作电压, 控制光线的通过量, 从而显示变化万千的图像。液晶本身并不会发光, 因此所有的液晶显示器都需要背光照明。背光灯管在液晶显示器打开的同时就一直被点亮。为了控制透光率, 把液晶单元放在两片偏振玻璃片之间。这样, 当液晶单元没有被加上电压的时候, 处于初始状态, 背光在通过时就会被液晶单元的特殊分子结构所极化, 光线被扭曲, 从而通过前面的偏振玻璃被人们所感知, 即产生“白色”效果。同理, 当液晶单元被加上电压之后, 它的分子结构会被改变, 这样光线的角度并不会被扭曲。于是光被显示器前面的偏振玻璃所阻隔, 无法被人们所感知, 即产生“黑色”效果。一般来说, 能显示的颜色越多, 越能显示复杂的图象, 画面的层次也更丰富。

显示屏是最直接的输出型人机界面。从简单的字符型显示屏到超高分辨率图形显示屏, 显示技术的不断提升在不停地改善着用户体验。除了分辨率之外, 颜色位数也是显示屏的重要参数之一, 通常为 6 位或 8 位。颜色位数用来表示显示屏能显示的每个颜色通道 (RGB) 颜色数量, 6 位屏可以显示颜色为 $2^{6 \times 3}$, 算下来 6 位屏最多可以显示 262 144 种颜色, 8 位屏最多可以显示 16 777 216 (2 的 24 次方) 种颜色。

目前 LCD 显示屏通常由 LCD 面板、驱动电路与相应的接口逻辑组成。LCD 面板的构造是在两片平行的玻璃基板当中放置液晶盒 (liquid crystal cell), 下基板玻璃上设置 TFT (Thin Film Transistor, 薄膜晶体管), 上基板玻璃上设置彩色滤光片, 通过 TFT 上的信号与电压改变来控制液晶分子的转动方向, 从而达到控制每个像素点偏振光射出与否, 最终达到显示目的。

显示控制器从系统存储器中读出要显示的图像数据，然后通过相应的显示接口输出到连接的显示器上。显示的图像以帧（frame）为单位存储在存储器中，该段存储空间称为帧缓冲（frame buffer）。每帧图像由像素（pixel）组成，像素有多种存储格式，最常见的像素格式为 ARGB（Alpha-Red-Green-Blue）格式，一个像素由 32 位数据表示，像素的不透明度以及各颜色通道（红、绿、蓝）依次由 8 位数据表示。显示控制器从存储器的帧缓冲空间读出图像数据，并按照输出接口对图像数据进行重新编码，输出显示。

通常，显示器上的图像需要不断刷新才能保持显示。刷新率多为 60Hz（即每秒显示控制器向显示屏传输 60 帧图像数据），这样才能获得较好的显示效果，人眼观察不会有闪烁感。因此，显示控制器需要按照相应的频率不断地读出帧缓冲中的图像数据，并按照一定的接口规范输出到显示器上。图 6-16 为嵌入式系统中图像数据的传输图。

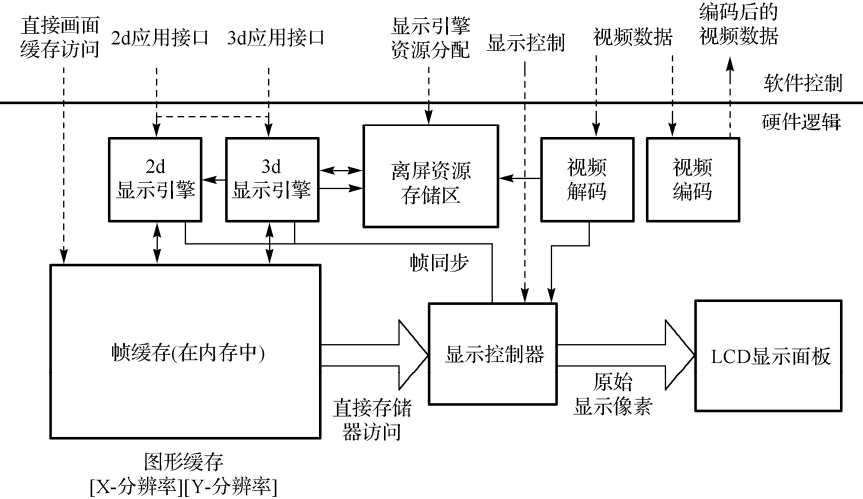


图 6-16 嵌入式系统中图像数据的传输图

帧缓冲中的数据有多种来源。处理器可以直接向帧缓冲中写入数据。嵌入式系统 SoC 多集成支持 2D/3D 加速功能的 GPU（Graphics Processing Unit，图形处理单元），通过 GPU 提供的命令，用户可以高效地更新帧缓冲中的图像数据（在没有 GPU 的系统中，所有的显示图像都是由 CPU 通过执行软件生成显示数据，并存储在帧缓存中。而对于有 GPU 的系统，程序员通过调用相应的 GPU 编程接口，利用 GPU 的硬件生成相应的图像数据，并写入相应的帧缓存）。此外，许多嵌入式微处理器还集成了 VPU（Video Processing Unit，视频处理单元）。VPU 可以用于视频解码，将压缩过的视频数据还原成一帧帧的图像数据。VPU 可以和显示控制配合工作，直接将图像数据按预定的视频帧率输出。VPU 也可以直接将解码后的图像数据写入帧缓冲，再由显示引擎读出显示。在某些应用中，视频解码后图像数据还需要进行一定的图像处理，如缩放、旋转等，此类操作通常称为视频后处理。后处理可以由专门的视频后处理模块进行（独立于 VPU 或集成于 VPU 中），也可以借助于 GPU 的 2D 加速功能进行。视频后处理前的原始图像数据，后处理过程中产生的临时图像数据，包括非视频解码应用时 GPU 进行图像处理的原始数据及临时数据，都需要分配相应的存储器空间，此类存储器空间通常称为离屏资源存储区（off-screen resources memory）。

显示控制器通过内置 DMA 直接从帧缓冲中读出图像数据。需要注意的是，显示控制器在读取某一帧图像时，不能更新该帧图像数据，否则显示的图像将出现上下部分不一致的问题，影响显示效果。通常的解决办法即设计多个帧缓冲区，进行乒乓操作，利用同步信号控制显示控制器切换读取的帧缓冲区。

通常系统需要显示多个来源的图像数据，如窗口内嵌视频画面。各个独立的图像帧来源称为显示层（display plane 或 Overlay）。显示层实际上是一块矩形图像，显示层的属性包括：图像帧数据首地址、像素数据格式、图像大小以及屏幕显示位置等，显示层在存储器中表现为一块属性相同的图像数据。显示系统需要根据配置拼合显示层，产生拼合后的图像帧，输出显示。拼合各显示层图像数据有两种方法。一种是使用显示控制器实现硬件拼合。窗口帧与视频帧具有不同的帧起始地址，且存储空间互不交叠。处理器、GPU、VPU 需要根据配置更新相应的图像帧数据。显示控制器同时读出所有有效的图像帧，按照设定进行叠加处理，最后拼合成显示图像经由显示接口输出，如图 6-17 所示。

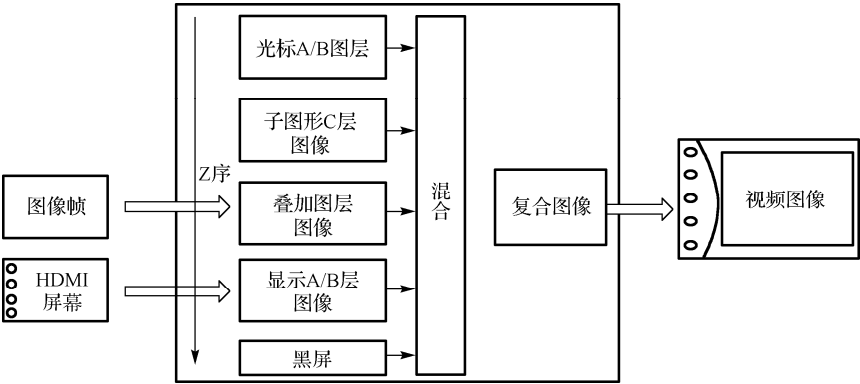


图 6-17 显示接口输出待显示数据的过程

除了使用显示控制器来实现多层图像的交叠显示外，目前嵌入式系统 SoC 逐渐开始使用 GPU 来进行显示层的拼合，GPU 根据显示层拼合顺序配置各显示层图像的不透明度，对有效显示层进行交叠操作，产生拼合后的显示图像帧，放入帧缓冲区，供显示控制器读取输出。

当然，使用显示控制器实现各显示层的硬件混合可以有效降低软件复杂度以及系统带宽需求。然而，对于 Android 等智能操作系统，用户界面由多个图层（图标、壁纸等）交叠产生，通常不同的显示控制器所支持的显示层可能并不相同，为了保证系统兼容性，Android 等操作系统多使用 CPU、GPU 混合用户界面的各个图层，作为显示控制器的一个显示层，送入帧缓冲区，输出显示。

混合操作通常采用透明度混合（alpha blending）的方式进行。显示层中的图像像素数据由透明度通道与颜色通道组成。例如，显示层 P1、P2 中的像素数据均为 ARGB 格式，透明度（A 通道）为 0xFF 代表该像素完全不透明，透明度为 0x00 代表该像素完全透明，显示层 P2 处于显示层 P1 之上，透明度混合之后显示像素的 RGB 值如图 6-18 所示。

$$\frac{(\text{显示层 P2 Alpha} \times \text{显示层 P2 RGB}) + ((0xFF - \text{显示层 P2 Alpha}) \times \text{显示层 P1 RGB})}{0xFF}$$

图 6-18 Alpha Blending 之后的显示像素 RGB 值的计算

随着显示信息的日益丰富以及人们对显示效果要求的逐渐提高，LCD 面板行业正快速地朝更高分辨率和每种颜色更多位数加速前进。只有提高主机到 LCD 面板的数据传输速率，分辨率和颜色数才能有所突破。如今大数据量的图像处理能力已经在主机端和显示端有很好的技术支持，处于主机端和显示设备端之间的信号传输线路和接口成为了显示技术发展的瓶颈和短板。为了满足数据量更大的图像显示，摆脱限制、进一步取得突破的重任便落在显示接口的身上。

接口逻辑接收显示数据，通过驱动电路控制 LCD 面板进行显示。整个显示系统如图 6-19 所示。

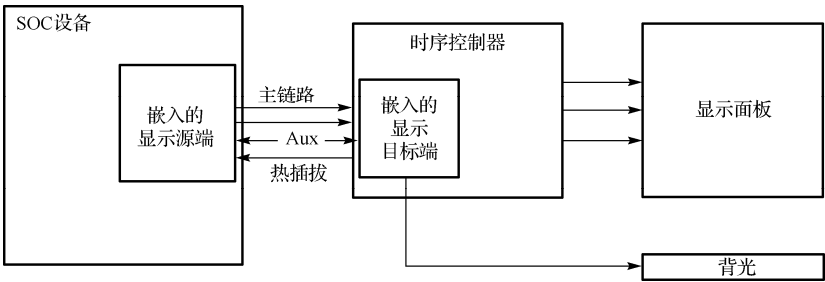


图 6-19 显示系统

下面介绍几种常用的 LCD 面板与驱动电路板之间的数字显示接口。

1) TTL 显示接口

当显示面板第一次面市时，就选择了经典的 TTL 数字接口作为标准。那时，面板尺寸不足 10 英寸，其 VGA 分辨率（640×480）为每个颜色 6 比特，带宽要求是 300Mbps。

TTL 信号是标准数字信号，很自然地可以驱动 TFT，控制 LCD 面板显示。6 位 TTL 接口线一共有 22 根（最少的，没有算地线和电源线），分别为 R、G、B 三基色信号（R0-R5、G0-G5、B0-B5），两个 HS、VS 行场同步信号，一个数据使能信号 DE，一个时钟信号 CLK。三基色信号是颜色信号；另外 4 根的信号（HS、VS、DE、CLK）是显示控制信号。图 6-20 所示为 TTL 接口信号。

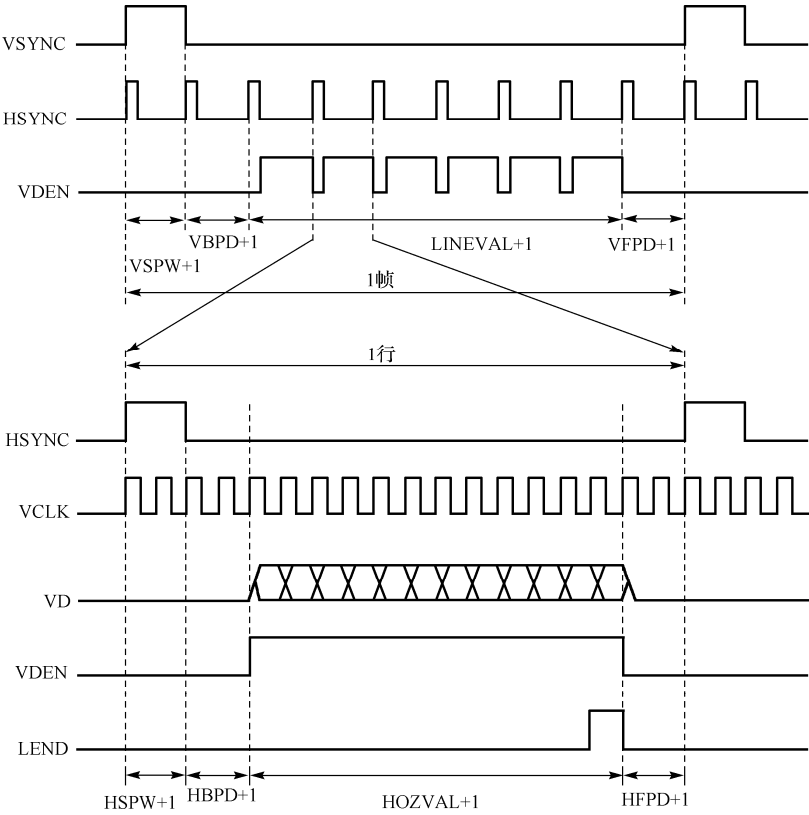


图 6-20 TTL 接口信号

图 6-21 显示了水平时序（一行的时序），包括水平同步脉冲和数据。HSYNC 的宽度以及 HSYNC 前后的延时都是可编程配置的。时序信号参数定义如下：

- H_WIDTH 定义了 HSYNC 脉冲的宽度，至少必须为 1。
- H_WAIT_2 定义了从 HSYNC 结束到 OE 脉冲开始之间的延时。
- H_WAIT_1 定义了从 OE 信号结束到 HSYNC 脉冲开始之间的延时。
- XMAX 定义了每行的像素数。

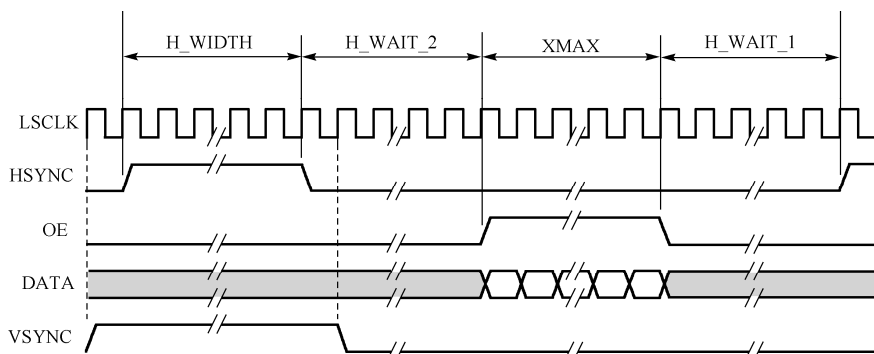


图 6-21 TFT 模式的水平同步脉冲时序

图 6-22 显示了垂直时序（一帧的时序）。一帧结束直到下一帧开始之间的延时是可编程的。时序信号参数是：

- V_WAIT_1 是以行数来衡量的延时。V_WAIT_1 = 1，在 VSYNC 之前有一个 HSYNC（时间为一个行周期）的延时。
- V_WIDTH 定义了 VSYNC 脉冲的宽度，V_WIDTH=1，VSYNC 包括一个 HSYNC 脉冲。V_WIDTH=2，VSYNC 包括两个 HSYNC 脉冲。
- V_WAIT_2 是以行数来衡量的延时。V_WAIT_2 = 1，在 VSYNC 之后有一个 HSYNC（时间为一个行周期）的延时。

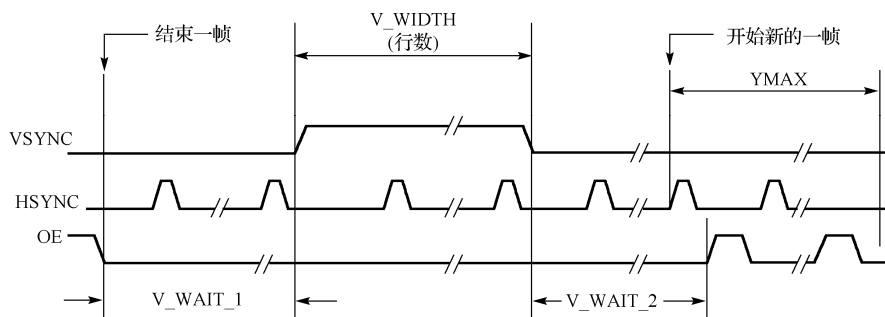


图 6-22 TFT 模式的垂直同步脉冲时序

图 6-21 和图 6-22 中接口信号管脚的意义如下：

- 显示屏在 SCLK 下降沿（部分显示屏为上升沿）时锁存数据。
- 水平同步脉冲 HSYNC 导致 TFT 屏开始新的一行。
- 垂直同步脉冲 VSYNC 导致 TFT 屏开始新的一帧。
- 输出使能 OE 作为输出图像信号的使能信号。
- 行数据 DATA 信号。

面板尺寸在 20 世纪 90 年代后半期增加到 15 英寸的范围，就要求采用 XGA 分辨率（1024×768），带宽需求跃升到了 850Mbps。由于 TTL 信号高电平有 3V 左右，对于高速率的长距离传输影响很大，

且抗干扰能力也比较差。因此，TTL 接口常用于板级连接，在嵌入式系统中较为常见。对于计算机平台，以及需要连接外部显示设备，在一定距离（数米）内传输图像数据的嵌入式系统来说，需要使用标准的显示传输接口。

VGA（Video Graphics Array，视频图形阵列），是 IBM 在 1987 年推出的使用模拟信号的一种视频传输标准，在当时具有分辨率高、显示速率快、颜色丰富等优点，在彩色显示器领域得到了广泛的应用。VGA 最早指的是 640×480 显示模式。随后 VESA（Video Electronics Standards Association，视频电子标准协会）在 VGA 基础上加以扩充，使其支持更高的分辨率，如 800×600 或 1024×768，这些扩充的模式就称为 Super VGA 模式，简称 SVGA。同时，标准还定义了相应的接口标准，即 VGA 接口（见图 6-23），也叫 D-Sub 接口，传输 R、G、B 模拟信号以及行、场同步信号。

当时 LCD 显示器尚未普及，CRT（Cathode Ray Tube，阴极射线管）显示器被广泛使用作为计算机的显示设备。CRT 的工作原理是电子束扫描成像，使用模拟信号控制成像，因此使用 VGA 接口将数字显示图像转换为模拟信号进行传输顺理成章，可以直接驱动 CRT 显示器。



图 6-23 VGA 接口

这个标准对于现今的个人电脑市场已经十分过时。即使如此，VGA 仍然是最多制造商所共同支持的一个标准，个人电脑在加载自己的独特驱动程序之前，都必须支持 VGA 的标准。例如，微软 Windows 系列产品的开机画面仍然使用 VGA 显示模式，这也说明其在显示标准中的重要性和兼容性。

LCD 面板直接由数字信号驱动，因此使用 VGA 接口时，驱动电路中需配置相应的 A/D（模数）转换器，将模拟信号转变为数字信号。同时使用 VGA 接口传输数据之前，数字显示图像使用 D/A 转换器转为模拟信号传输。这样，经过两次 D/A、A/D 转换后，不可避免地造成一些图像细节的损失。VGA 接口应用于 CRT 显示器无可厚非，但用于连接 LCD 显示器，转换过程的图像损失会使显示效果略微下降。

2)* DVI 显示接口

DVI 接口是 1999 年由 Silicon Image、Intel、Compaq、IBM、HP、NEC、Fujitsu 等公司共同组成的 DDWG（Digital Display Working Group，数字显示工作组）推出的接口标准，其外观是一个 24 针的接插件。

DVI 接口有多种规格，分为 DVI-A、DVI-D 和 DVI-I，它是以 Silicon Image 公司的 PanalLink 接口技术为基础，基于 TMDS（Transition Minimized Differential Signaling，最小化传输差分信号）标准传输显示数据。TMDS 是一种差分信号传输机制，可以将像素数据编码，并通过串行连接传递。一个 DVI 显示系统包括一个发送器和一个接收器。显示图像的数字信号由发送器按照 TMDS 协议编码后通过 TMDS 通道发送给接收器，经过解码送给数字显示设备，还原为显示图像。

DVI 接口包括 4 对双绞线：红、绿、蓝及时钟。红、绿、蓝 3 对双绞线以比特流的形式传输 24 位像素值。DVI 接口有 DVI1.0 和 DVI2.0 两种标准，其中 DVI1.0 仅用了其中的一组信号传输信道，传输图像的最高像素时钟为 165M（1600×1200@60Hz,UXGA），信道中的最高信号传输码流为 1.65Gbps。DVI2.0 则用了全部的两组信号传输信道，传输图像的最高像素时钟为 330M，每组信道中的最高信号传输码流也为 1.65Gbps。DVI 接口各个引脚的定义如图 6-24 所示。

HDMI（High Definition Multimedia Interface，高分辨率多媒体接口）接口是基于 DVI 接口制定的，可以看作是 DVI 接口的强化与延伸，两者可以兼容。HDMI 接口在保证高品质的情况下能够以数码形式传输未经压缩的高分辨率视频和多声道音频数据。HDMI 接口不仅能够支持 1080p 高清分辨率显示，还可以支持最先进的数字音频格式，而且只用一条 HDMI 线连接，无需数字音频连线。与 DVI 接口相比，HDMI 接口可以同时传输音频及视频信号，且接口体积小，其灵活性和方便性比较有优势。HDMI

接口规范定义了 4 种 HDMI 接头，即 HDMI A~D。HDMI A 为最常见的 HDMI 接头；HDMI D 接头又称为 micro HDMI 接头，尺寸为 2.8mm×6.4mm，主要应用在一些小型的移动设备上，如智能手机、平板电脑等。如图 6-25 所示，左侧为 HDMI D 接口，右侧为 HDMI A 接口。通过 HDMI D-HDMI A 接口转换，移动设备可以将画面输出到带有 HDMI 接口的外接显示器。



图 6-24 DVI 显示接口

(1) HDMI 接口在保持高品质的情况下能够以数字形式传输未经压缩的高分辨率视频和多声道音频数据，其卓越性能超越了以往所有的产品。



图 6-25 HDMI 接口

(2) HDMI 接口采用单线连接，可以同时传输视频信号与音频信号。

(3) HDMI 接口的线缆没有长度限制。DVI 接口的线缆长度不能超过 8 米，否则将影响画面质量，而符合 HDMI 接口标准的产品则没有这个问题。

(4) HDMI 接口标准可搭配 HDCP (High-Bandwidth Digital Content Protection，高带宽数码内容保护)，以防止具有著作权的影音内容遭到未经授权的复制。

DVI 接口和 HDMI 接口都是串行传输，采用差分信号来降低 EMI，提高信号传输速率。DVI 接口及 HDMI 接口采用更高级的编码算法，可以进一步降低 EMI，从而使得接收端具有更健壮的时钟恢复性能。DVI 接口已成功应用于计算机领域，HDMI 接口也成功推向了消费电子市场。

表 6-3 为几种常见数字显示接口的对比表。

表 6-3 几种常见数字显示接口的对比

	DisplayPort	LVDS (双通道)	DVI (单通道)	HDMI
数据对	1, 2 或 4 (可选)	8	3	3
时钟对	None (嵌入的)	2	1	1
每对的比特率	2.7 或 1.62Gb/s (可扩展)	945Mb/s (135MHz clock)	Max 1.65Gb/s	Max 2.25Gb/s
总的原始带宽	10.8Gb/s	7.56Gb/s	4.95Gb/s	6.75Gb/s
颜色深度 (每色的比特数)	6, 8, 10, 12, 16	8	8	8, 10, 12, 16
音频支持	Yes	No	No	Yes
信道编码	ANSI8B/10B	None	TMDS	TMDS
Aux 通道	1Mbps Aux CH	None	DDC	DDC
内容保护	HDCP (可选)	None	HDCP (可选)	HDCP (必选)
接口类型	外部和内部, AC-coupled	Internal, DC-coupled	External, DC-coupled	External, DC-coupled or AC-coupled

2. SEP4020 中的 LCDC 控制器

LCDC 是连接在 AHB 总线上的 AMBA master-slave 模块，在不需要 ARM7TDMI 的参与下能够独自提供给多种 LCD 显示板所必需的数字接口信号，包括黑白、灰度、彩色液晶屏。

LCDC 基本功能如下：

- 兼容 AMBA2.0 规范，是连接在 AHB 总线上的 AMBA master-slave 模块。
- 显示模式：支持黑白屏；支持 4 级和 16 级灰度的单色 STN 屏；支持最高 65536 色的彩色 TFT 屏。
- 分辨率可配，最大支持 800×600 的分辨率，推荐使用 320×240 的分辨率。
- 内部 20 级 4 比特的调色板，调色板软件可配置。
- 可编程控制的 AC 偏置信号。
- 像素时钟由系统主频分频。
- 使用内嵌的 DMA 方式进行取数据操作。
- 深度为 16、宽度为 32 的 FIFO 用于缓存显示数据。
- 大小印第安格式软件可配置。
- 每行的开始和结束等待时间软件可配。
- 每帧的开始和结束等待时间软件可配。
- 帧脉冲、行脉冲、像素时钟、像素信号和输出使能信号的极性软件可配。
- 可编程的 FIFO 下溢中断。

LCDC 的结构框图如图 6-26 所示。

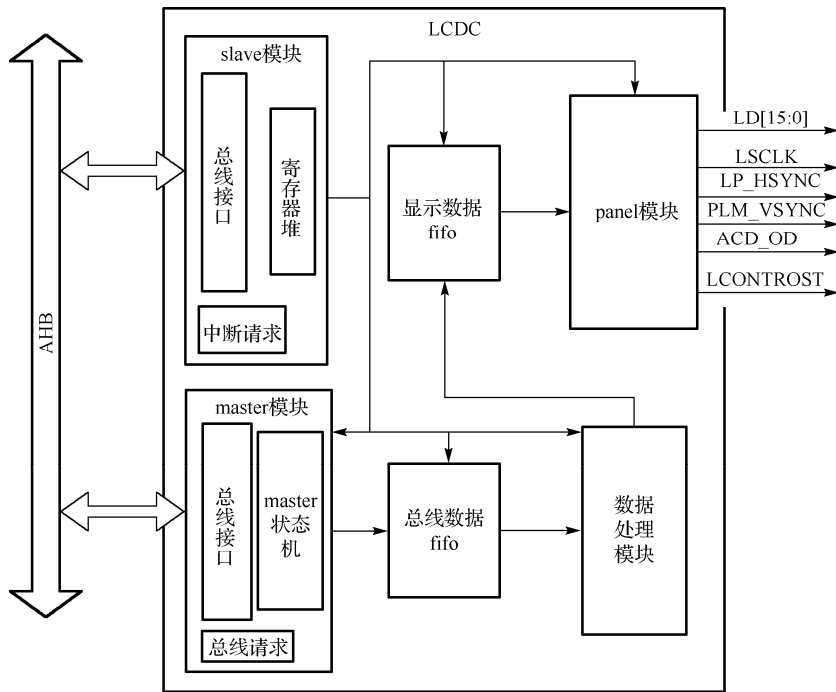


图 6-26 LCDC 的结构框图

3. LCDC 驱动

下面先来看看 SEP4020 中的 LCDC 控制器中的寄存器。

① 屏幕起始地址寄存器 SSA: 用于存放每帧数据的第一个字在存储器中的地址, 即 Framebuffer 的地址。当用户配置完 LCDC 并使能后, 以及每读完一帧数据后, LCDC 将以本寄存器的值为地址读取相应存储器单元的数据作为一帧数据中的第一个字, 然后以递增地址的方式读取一帧中剩余的数据。这个地址必须是开始在 4 字节的边界, 确保能存放一幅完整的图像。

② 屏幕尺寸寄存器 SIZE: 确定屏幕尺寸, 屏幕尺寸由屏幕宽度和屏幕高度组成, 用户可以根据实际硬件的屏幕大小来对该寄存器进行配置。

③ 面板配置寄存器 PCR: 这个寄存器比较重要, 它决定了 TFT 显示的接口, 彩色显示接口, Panel 总线宽度, 像素位数, 像素极性, 首行标志极性, 行脉冲极性, LCD 移位时钟极性, 输出使能极性, LSCLK 空闲使能, 印第安格式选择, ACD 时钟源选择, 晶向变换, LSCLK 选择, 像素时钟间隔。其中, 对刷新率起作用的是总线宽度、像素位数和像素时钟间隔。

④ 水平配置寄存器 HCR: 确定每刷新一行所需要的等待时间。

⑤ 垂直配置寄存器 VCR: 确定每刷新一帧所需要的等待时间。

⑥ PWM 对比度控制寄存器 PWMR: 确定屏幕的对比度。

⑦ 使能控制寄存器 LECR: 用于使能或禁止 LCDC。

⑧ DMACR 控制寄存器: 确定 LCD 控制器自带的 DMA 的传输方式和大小。

⑨ LCDICR 中断使能寄存器: 用于使能或屏蔽 LCDC 模块输出的中断信号。

⑩ LCDISR 中断状态寄存器: 存放了 LCDC 的中断状态标志。

⑪ 灰度调色映射寄存器组 LGPMR: 用于设置灰度显示屏的各级灰度。

在 1bpp (1bit/pixel) 情况下, LCD 屏宽设置实例如图 6-27 所示。假设 LCD 屏幕的大小为 240×320, 每个像素使用 1 比特表示, 则实际配置的大小可以为 256×320 (其中高度 256 为 32 的倍数)。此时, 屏幕尺寸寄存器(SIZE)中参数的 XMAX = 16, YMAX = 320, 在图中可以看到, 每行的最后 16 个像素不会显示在 LCD 屏幕上。

下面分别就 Framebuffer、刷新率的配置以及彩色 TFT 屏的初始化配置做一介绍。

(1) Framebuffer。

Framebuffer 又称为显示缓冲区, 是存放需要显示的像素数据的一段连续的存储器空间, 而且从首地址表示的第一个存储单元开始, 是和物理屏幕上像素点一一对应的。比如, 我们要把帧 BUFFER 放在 0x31200000 开始的区域, 则只要写语句* (RP) SSA = 0x31200000; 其中* (RP) 是一个寄存器写指针的宏定义。那么, 如果我们是采用 16 位彩色显示, 则 0x31200000 和 0x31200001 两个单元中的 16 位数据就与屏幕左上第一行第一个像素点对应; 相应地, 0x31200002 和 0x31200003 两个单元中的 16 位数据就与屏幕左上第一行第二个像素点对应; 以此类推。当然, 物理屏幕扫描顺序决定了屏幕上的像素点顺序, 因此, 了解所用显示屏的扫描顺序也是必要的。假设使用的屏幕是 240×320 (X 方向为 240, Y 方向为 320, 单位为像素点), 写语句* (RP) SIZE = 0x00f00140 即可。

(2) 刷新率的配置。

它主要由液晶屏的尺寸、PCR 寄存器中的总线宽度和像素时钟间隔、HCR 寄存器中的水平同步脉冲宽度与行与行之间的等待时间宽度来共同作用。用 X 表示液晶屏的宽度, Y 表示液晶屏的高度, PBSIZE 表示总线宽度, PCD 表示像素时钟间隔, H_WIDTH 表示水平同步脉冲宽度, H_WAIT_1 表

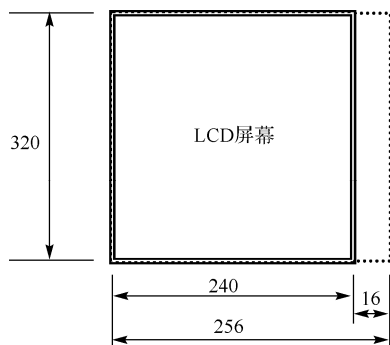


图 6-27 1bpp 情况下 LCD 屏参数配置实例

示 HSYNC/LP 和上一行结束的等待时间，H_WAIT_2 表示 HSYNC/LP 和下一行开始前的等待时间。那么，刷新率的计算公式如下：

$$1/f = ((X + H_WIDTH + H_WAIT_1 + H_WAIT_2) \times Y / PBSIZE) \times PCD / Hz$$

其中，f 是刷新率，Hz 是总线的频率，也就是工作频率。

(3) 彩色屏 TFT 屏的初始化配置。

- 配置 LCD 数据帧的起始地址。
- 屏幕尺寸设定为 240×320。
- LCD 被动模式，单色显示，4 比特总线宽度，2bpp，像素极性为高电平有效，小印第安格式，最后刷新率配置必须写奇数。
- 选取像素时钟，在对比度控制寄存器中关闭对比度控制，最后 7 位设置输出脉冲数目。
- Burst=1 设置 DMA 高标志 7 和低标志 3。
- LCD 数据帧的起始地址。
- 使能 LCDC。
- 中断使能。

6.3.2 音频接口

1. 音频接口概述

音频数据的采集、处理和传输是多媒体技术的重要组成部分。众多的数字音频系统已经进入消费市场，对于设备生产厂商来说，标准化的信息传输接口可以有效提高设备的通用性。音频信号为模拟信号，大部分数字音频系统采用脉冲编码调制（PCM，Pulse-Code Modulation）技术进行模拟信号的数字化。PCM 依照相同间距将模拟信号分成数段，然后用独特的数码记号（通常是二进制）来量化模拟信号的幅度。

图 6-28 为模拟信号进行 4 位 PCM 转换的采样与量化过程。一个正弦波（红色曲线）被取样和量化为 PCM。正弦波每隔一段固定时间进行一次采样，即 X 轴的刻度。每一个样本则依照某种运算方法（在这个例子中为上限函数），选定它们在 Y 轴上的位置。这样便产生完全离散的输入信号，很容易编码数字化。以图 6-28 为例，可以很清楚地看出样本为 9、11、12、13、14、14、15、15、15、14、…、等，将它们以二进制编码，就得到一组一组的数字：1001、1011、1100、1101、1110、1111、1111、1111、1110、…、等，这些数码数据之后就可以保存下来，由处理器进行相应操作。

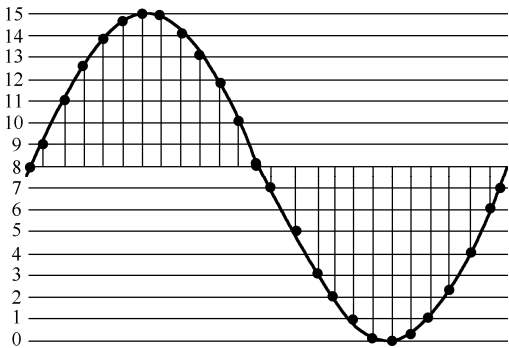


图 6-28 模拟信号进行 4 位 PCM 转换的采样与量化过程

采用 PCM 技术数字化的音频信号还应包含相应的采样信息。为了实现不同数字音频设备之间的

数字化音频信号传输，业界定义了相关的传输规范。嵌入式系统常见的传输接口有 PCM 接口与 I2S (Inter-IC Sound, 芯片间音频) 接口。

1) PCM 接口

最简单的音频接口是 PCM (脉冲编码调制) 接口。PCM 接口通常用于语音信号传输 (采样频率为 8kHz 的窄带模式以及采样频率为 16kHz 的宽带模式)，在嵌入式系统中常用于蓝牙设备及调制解调器的连接。

PCM 接口信号由时钟脉冲 (CLK)、帧同步信号 (FS)、接收数据 (DR) 和发送数据 (DX) 组成。FS 信号频率等于采样率，FS 信号之后开始传输采样数据，每次传输一个规定的字长。采样数据按位在 FS 信号的上升沿进行传输，由最高位开始，1 个时钟周期传输 1 位数据。采样位数可以小于字长，不足的位数由 0 补满；采样位数也可以大于字长，多余的位数放弃传输。

PCM 接口可以传输单声道音频数据，也可以传输立体声音频数据，右声道数据字紧接左声道数据字。进行立体声音频数据传输时，PCM 接口传输信号时序如图 6-29 所示。

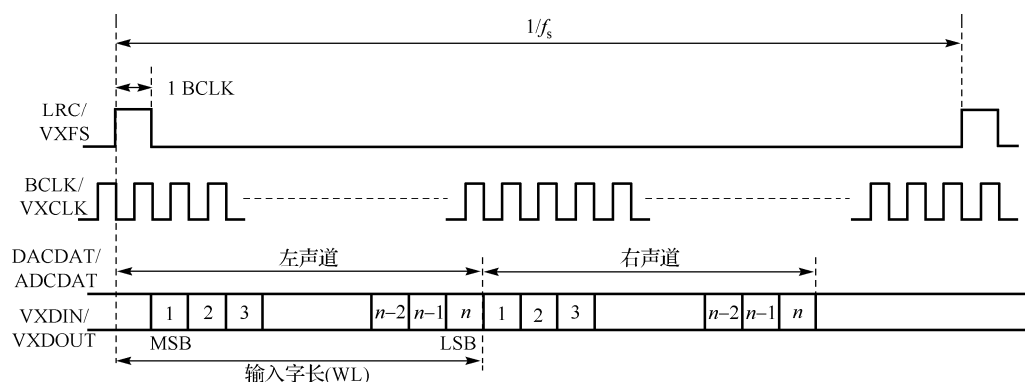


图 6-29 PCM 接口信号时序

2) I2S 接口

I2S (Inter-IC Sound, 芯片间音频) 是常用的音频接口标准，用于数字音频设备，如 CD 机、多媒体处理器等。I2S 属于同步串行接口，时钟信号和数字信号分离，数据在时钟的驱动下串行传输。该接口是飞利浦公司为数字音频设备之间的音频数据传输而制定的一种接口标准。I2S 接口主要用于传输采样频率为 8~48kHz 的音频数据，采样位数为 16~32 位。

在飞利浦公司的 I2S 标准中，既规定了硬件接口规范，也规定了数字音频数据的格式。I2S 有 3 个主要信号：

- 位时钟信号 BCLK，也称为串行时钟信号 SCLK，即对应数字音频的每一位数据，BCLK 的频率 = $2 \times \text{采样频率} \times \text{采样位数}$ 。
- 左右声道信号 LRC，也称为字段选择信号 WS，用于切换左右声道的数据。LRC 为高电平表示正在传输的是右声道的数据，为低电平则表示正在传输的是左声道的数据。LRC 信号的频率等于采样频率。
- 串行数据信号 SDATA，就是用二进制补码表示的音频数据。

一个典型的 I2S 传输如图 6-30 所示。

在统一的 I2S 接口下，存在多种不同的数据格式。根据 SDATA 数据相对于 LRC 和 BCLK 的位置不同，分为左对齐 (较少使用)、I2S 格式 (即飞利浦规定的格式) 和右对齐 (也叫日本格式、普通格式)。非 I2S 格式及 I2S 格式的数据传输如图 6-31 所示。

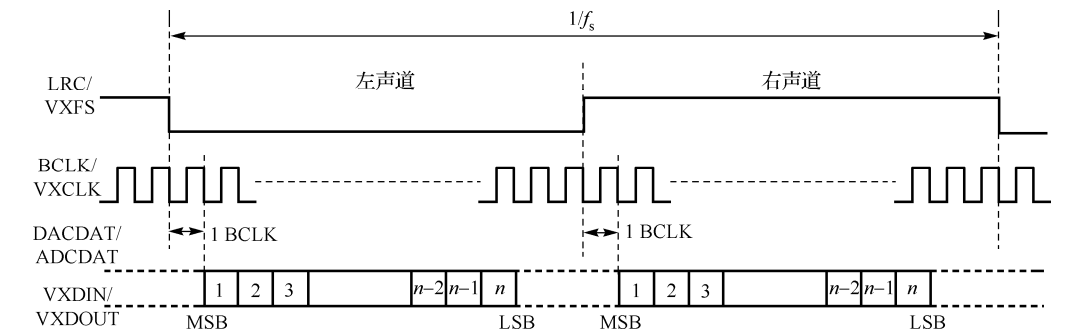


图 6-30 I2S 接口信号时序

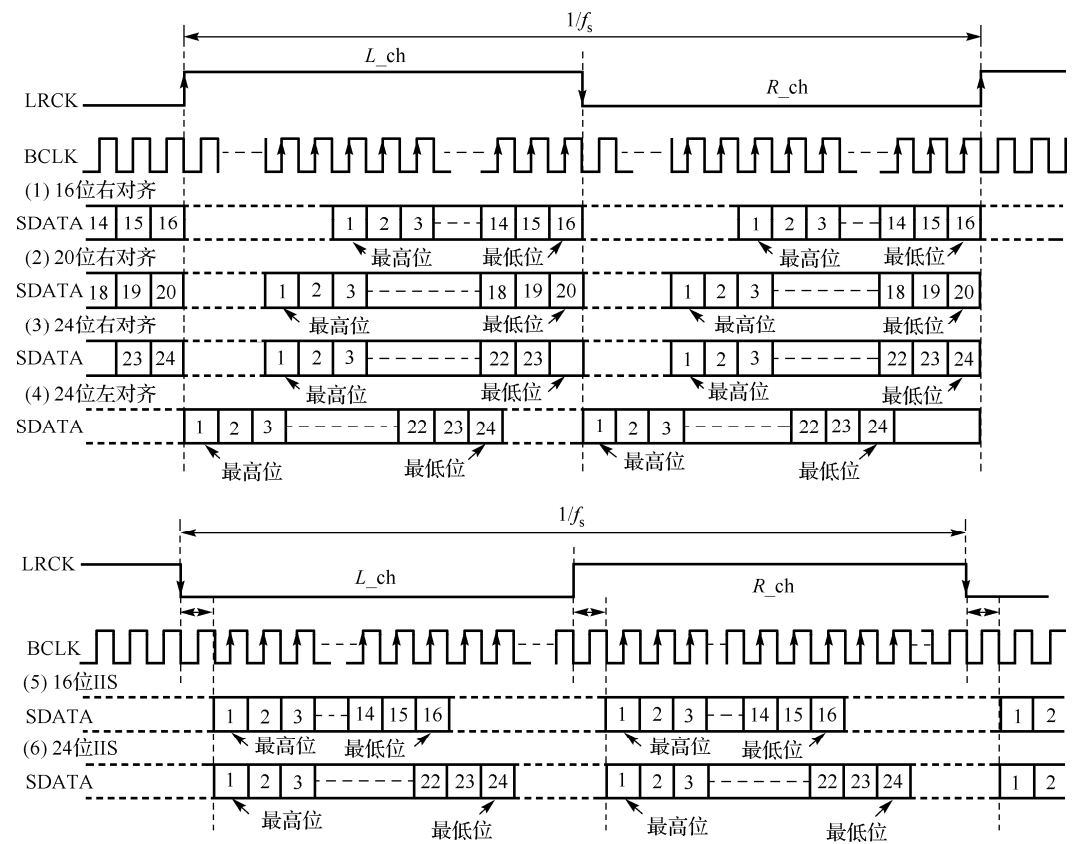


图 6-31 非 I2S 格式和 I2S 格式的数据传输比较

同 PCM 接口一样，I2S 传输数据的采样位数可以小于字长，不足的位数由 0 补满；采样位数也可以大于字长，多余的位数放弃传输。

通常来说，为了保证数字音频信号的正确传输，发送端和接收端应该采用相同的数据格式和长度。然而，从图 6-31 可以看出，对于 I2S 传输来说，不论何种数据格式，数据长度都可以不同。I2S 格式的数据信号无论有多少位有效数据，数据的最高位总是出现在 LRC 变化（也就是一帧开始）后的第 2 个 BCLK 脉冲处，这就使得接收端与发送端的有效位数可以不同。如果接收端能处理的有效位数少于发送端，则可以放弃数据帧中多余的低位数据；如果接收端能处理的有效位数多于发送端，则可以自行补足剩余的位。这种同步机制使得数字音频设备的互联更加方便，而且不会造成数据错位。

2. SEP4020 中的 I2S 控制器

SEP4020 嵌入式微处理器中的 I2S 控制器主要功能包括:

- 支持 MASTER 和 SLAVE 模式。
- 支持 TRANSMITTER 和 RECEIVER 功能。
- 支持 32 位、16 位、8 位音频数据字长。
- 支持立体声和单声道。
- 支持静音和停止播放。
- 数据高位 (MSB) 先出/先入。
- 接收发送共享 4×32 数据 FIFO。
- 支持 DMA 传输模式。

I2S 工作模式分为 MASTER 模式和 SLAVE 模式。MASTER 模式下, I2S 向外输出时钟; SLAVE 模式下, 时钟由外部芯片提供。常见的 SoC 芯片的 I2S 工作在 MASTER 模式, 处理器芯片外接音频 CODEC (音频的编解码芯片, 也就是 AD/DA 转换芯片), SoC 芯片向外提供数据, CODEC 做 DA 转换发出声音, 录音则是 CODEC 提供 AD 转换后的数据, SoC 芯片把数据接收到 MEMORY 中。无论在 MASTER 模式还是在 SLAVE 模式, I2S 控制器都可以作为数据发送方 (Transmitter) 或者数据接收方 (Receiver), 如果 I2S 控制器向外接 CODEC 提供数据, 则 I2S 控制器是数据发送方 (Transmitter), 如果 I2S 控制器从外接 CODEC 接收数据, 这时 I2S 控制器是数据接收方 (Receiver)。

I2S 的接口由 3 个信号组成, SCK 是驱动时钟, WS 是字选择信号, SD 是数据, RECEIVER 在 SCK 的上升沿采样 WS 和 SD 信号。SCK 由总线时钟分频而来, 分频比可配。WS 是占空比为 50% 的周期性信号, 高低电平分别对应左右 (或者右左) 声道, 周期长度与音频数据的字长有关, 例如, 8 位的双声道音频数据, 对应的 WS 周期就是 16 个 SCK, 8 个左声道数据, 8 个右声道数据; 若是单声道, WS 的高低电平将使用相同的数据。左右声道对应的 WS 信号电平可配, 可以左低右高或者左高右低。I2S 信号传输波形如图 6-32 所示。

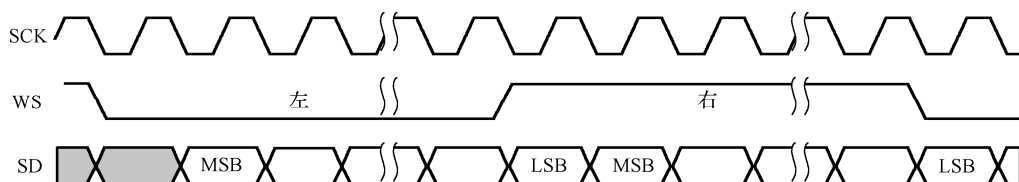


图 6-32 I2S 信号传输波形 (先左)

I2S 的数据 (SD 信号) 的变化比字选择 (WS) 晚一个时钟。I2S 支持 8 位、16 位和 32 位字传输, 数据的传输支持 DMA 模式。在单声道模式下, 左右声道输出同样的数据。当 I2S 作为 TRANSMITTER 工作时, 数据从总线传输到 I2S 的数据 FIFO, 然后根据设定的字长、声道将数据送出, 数据的高位先出。当 FIFO 为半空的时候, 发出中断和 DMA 请求; 当 FIFO 为空时, 发出空中断, I2S 将继续重复发送最后一个数据 (这种情况下, CODEC 将会发出不是用户想要的声音), 直到有新的数据写到 FIFO 中。当 I2S 作为 RECEIVER 工作时, 数据从外接的 CODEC 输入到 FIFO, 当 FIFO 半满时发出中断和 DMA 请求, 当 FIFO 为满时, 发出满中断, I2S 将继续接收数据, 但之后接收的数据会被丢弃而不被存入 FIFO 中。除了工作在标准的 I2S 模式, I2S 模块还可以在 WS 和 SD 对齐的方式下工作。

SEP4020 中 I2S 控制器设计了下列寄存器。

- I2S 控制寄存器: SCK 时钟分频参数 (MASTER 模式下使用)、双声道/单声道、MASTER 模式/SLAVE 模式、TRANSMITTER/RECEIVER、音频数据位宽等的配置。

- I2S 数据寄存器：I2S 控制器中 FIFO 的端口。
- I2S 中断寄存器：中断是否使能。
- I2S 状态寄存器：I2S 控制器中 FIFO 的状态。

I2S 控制器的结构框图如图 6-33 所示。

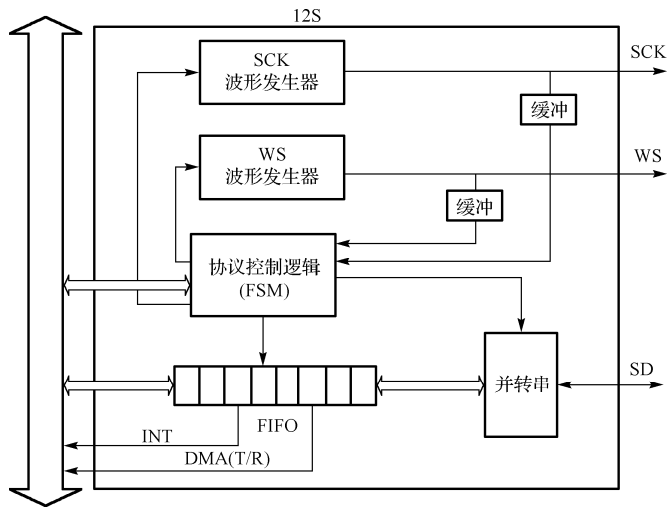


图 6-33 I2S 控制器结构框图

3. I2S 驱动

SEP4020 处理器的 I2S 控制器模块不是独立工作的，需要和芯片中的 PWM 模块和 DMA 模块协同工作。

SYSCLK 设置：CODEC 的系统时钟 SYSCLK 是由 PWM 输入的，通过配置 PWM 模块以 PWM 模式输出固定周期和占空比为 50% 的波形。数据寄存器的值设为周期寄存器的值的一半。

DMA 设置：PCM 数据加载到内存，通过 DMA 传到 I2S_DATA 中，再由 I2S 输入到 CODEC 中播放。DMA 一次最多能传输 2K 数据 (byte, halfword, word)，为了能连续传输，可以在 DMA 传输中使能传输完成中断，然后在中断处理函数中再设置 DMA 通道信息，继续传输。也可以用 DMA 链表方式进行连续传输，这样则无需用到中断。

I2S 设置：配置信息写入 CODEC 以后，配置 I2S 模块的寄存器，主要是注意配置 I2S 控制寄存器中的 SCK 分频参数。

对 96M 的主频，16 位的音频数据，44.1K 采样率，I2S 的 SCK 分频参数 DIV（将总线时钟进行 $(DIV+1) \times 2$ 分频）一般设为 0x1f， $96M / ((0x1f+1) \times 2) = 1500K$ ， $1500/32=46.875K$ ，当 CODEC 的 SYSCLK 设为 $96M/8=12M$ 时，I2S 的 SCK 分频参数配置为 0x1f 时效果最佳。

22K 采样率，SCK 分频参数 DIV 设为 0x3f。

11K 采样率，SCK 分频参数 DIV 设为 0x7f。

8K 采样率，SCK 分频参数 DIV 设为 0xaf。

6.3.3 触摸屏接口

1. 触摸屏原理

为了操作上的方便，人们使用触摸屏代替按钮、鼠标或键盘等输入设备来控制嵌入式系统。触摸

屏又称为触控面板，是可以识别用户触摸操作的感应式输入装置。

早在 1971 年，美国人 Sam Hurt 就发明了触摸屏技术。20 世纪 90 年代触摸屏技术才真正进入日常生活。触摸屏作为一种最新的输入设备，是目前最简单、方便、自然的输入设备。触摸屏具有坚固耐用、反应速度快、节省空间、易于交流等许多优点。用户只要用手指轻轻地触碰显示屏上的图标或文字就能实现对主机的操作，从而使人机交互更为直截了当，这种技术极大方便了那些不懂电脑操作的用户。这种人机交互方式赋予了多媒体以崭新的面貌，是极富吸引力的全新多媒体交互设备。

触摸屏的应用范围非常广泛，主要有公共信息的查询，如电信局、税务局、银行、电力等部门的业务查询；城市街头的信息查询；此外还可广泛应用于工业控制、军事指挥、电子游戏、点歌点菜、多媒体教学等领域。

1) 触摸屏的工作原理

触摸屏通常安装在显示屏表面，用于检测用户触摸位置。触摸屏控制器的主要作用是控制触摸屏，识别触摸信息，并转换成触点坐标送给处理器。通常触摸屏、触摸屏控制器与显示屏搭配使用，合称为触摸显示屏。触摸显示屏实现了输入和输出的合二为一，用户使用手指或其他物体接触屏幕显示内容，系统将根据手指触摸位置来选择信息输入。触摸屏的工作原理如图 6-34 所示。

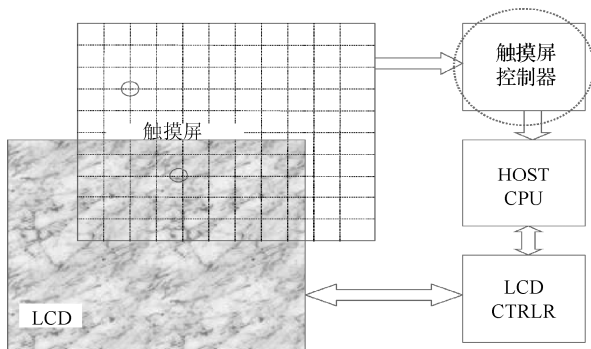


图 6-34 触摸屏的工作原理

2) 触摸屏的主要类型

按照触摸屏的工作原理和传输信息的介质，触摸屏可分为 4 种，分别为电阻式、红外线式、电容感应式以及表面声波式。目前，嵌入式系统普遍采用的触摸屏为电阻式触摸屏及电容式触摸屏。

(1) 电阻式触摸屏。

电阻式触摸屏基本上是薄膜加上玻璃的结构，由一层玻璃或有机玻璃作为基层，表面涂有一层透明的阻性层（OTI，纳米铟锡金属氧化物），上面再盖有一层外表面硬化处理、光滑防刮的塑料层，它的内表面也涂有一层 OTI，在两层阻性层之间有许多细小、有弹性的透明隔离点将两层隔开绝缘。其中一面阻性层接通 Y 轴方向的均匀电压 VDD，当触摸屏表面受到的压力（如通过笔尖或手指进行按压）足够大时，两层导电层出现一个接触点，使得另一面阻性层（即侦测层）的电压由零变为非零。控制器检测到侦测层的电压变化，进行 A/D 转换，并将得到的电压值与 VDD 相比，即可得触摸点的 Y 轴坐标，同理得出 X 轴的坐标。这就是电阻技术触摸屏共同的最基本原理。

电阻式触摸屏的 OTI 涂层比较薄且容易脆断，涂得太厚又会降低透光且形成内反射降低清晰度，OTI 外虽多加了一层薄塑料保护层，但依然容易被锐利物件所破坏；且由于经常被触动，表层 OTI 使用一定时间后会出现细小裂纹，甚至变形，如其中一点的外层 OTI 受破坏而断裂，便失去作为导电体的作用，触摸屏的寿命并不长久。但电阻式触摸屏不受尘埃、水、污物的影响。

电阻屏根据引出线数多少，分为四线、五线等多线电阻触摸屏。

① 四线触摸屏。

四线触摸屏包含两个阻性层。其中一层在屏幕的左右边缘各有一条垂直总线，另一层在屏幕的底部和顶部各有一条水平总线，如图 6-36 所示。为了在 X 轴方向进行测量，将左侧总线偏置为 0V，右侧总线偏置为 VDD。将顶部或底部总线连接到 ADC，当顶层和底层相接触时即可进行一次测量。为了在 Y 轴方向进行测量，将顶部总线偏置为 VDD，底部总线偏置为 0V。将 ADC 输入端接左侧总线或右侧总线，当顶层与底层相接触时即可对电压进行测量。

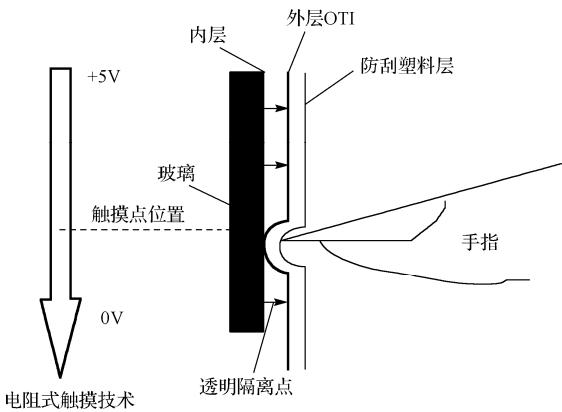


图 6-35 电阻式触摸屏

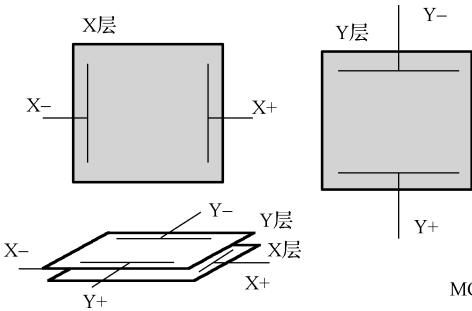


图 6-36 四线触摸屏

② 五线触摸屏。

五线触摸屏使用了一个阻性层和一个导电层。导电层有一个触点，通常在其一侧的边缘。阻性层的四个角上各有一个触点。为了在 X 轴方向进行测量，将左上角和左下角偏置到 VDD，右上角和右下角接地。由于左、右角为同一电压，其效果与连接左右侧的总线差不多，类似于四线触摸屏中采用的方法。为了沿 Y 轴方向进行测量，将左上角和右上角偏置为 VDD，左下角和右下角偏置为 0V。由于上、下角分别为同一电压，其效果与连接顶部和底部边缘的总线大致相同，类似于在四线触摸屏中采用的方法。

相比四线电阻屏，五线电阻触摸屏的内层是导电玻璃而不是导电涂层，导电玻璃的工艺使得内层的寿命得到极大的提高，并且可以提高透光率。五线电阻屏把工作面的任务都交给寿命长的内层，而外层只用来作为导体，并且采用了延展性好、电阻率低的镍金透明导电层，极大地延长了外层寿命。五线电阻触摸屏可以通过精密的电阻网络来校正内层的线性问题，补偿工作面有可能产生的线性失真。五线电阻屏可以说是目前最好的电阻触摸屏。

③ 七线触摸屏。

七线触摸屏的实现方法除了在左上角和右下角各增加一根线之外，与五线触摸屏相同。执行屏幕测量时，将左上角的一根线连到 VDD，另一根线接 ADC 的正参考端。同时，右下角的一根线接 0V，另一根线连接 ADC 的负参考端。导电层仍用来测量分压器的电压。

④ 八线触摸屏。

除了在每条总线上各增加一根线之外，八线触摸屏的实现方法与四线触摸屏相同。对于 VDD 总线，将一根线用来连接 VDD，另一根线作为 ADC 的数模转换器的正参考输入。对于 0V 总线，将一根线用来连接 0V，另一根线作为 ADC 的数模转换器的负参考输入。未偏置层上的 4 根线中，任何一根都可用来测量分压器的电压。

(2) 电容式触摸屏。

电阻式触摸屏为单点式触摸屏，只支持单点触控。2007年，苹果公司推出 iPhone 系列智能手机，使用电容式触摸屏，支持多点触控技术，迅速获得了全球大众的青睐，给用户带来独特而又神奇的应用体验，使得支持多点触控的电容式触摸屏大放异彩。

电容式触摸屏的构造主要是在玻璃屏幕上镀一层透明的薄膜体层，再在导体层外加上一块保护玻璃，双玻璃设计能彻底保护导体层及感应器。电容触摸屏的双玻璃不但能保护触摸屏结构，更有效地防止外在环境因素对触摸屏造成影响，就算屏幕沾有污秽、尘埃或油渍，电容式触摸屏依然能准确算出触摸位置。

早期的电容式触摸屏为表面式电容触摸屏。在表面式电容触摸屏四边均镀上狭长的电极，在导体层内形成一个低电压交流电场。用户触摸屏幕时，由于人体电场，手指与导体层间会形成一个耦合电容，四边电极发出的电流会流向触点，而电流强弱与手指到电极的距离成正比，位于触摸屏幕后的控制器便会计算电流强弱，得到触摸点的位置，见图 6-37。这种技术很长一段时间被用在信息亭触摸屏上。

但是表面电容式触摸屏有一些局限性，它只能识别一个手指或者一次触摸。另外，考虑到电极的尺寸，对于小尺寸屏幕（如那些用在手持式平台上的屏幕）是不切实际的。

投射电容式触摸屏（projected-capacitive touch screen）利用触摸屏电极发射出的静电场线，可分为自电容屏和互电容屏两种类型。

在玻璃表面用 ITO（一种透明的导电材料）制作成横向与纵向电极阵列，这些横向和纵向的电极分别与地构成电容，这个电容就是通常所说的自电容，也就是电极对地的电容。当手指触摸到电容屏时，手指的电容将会叠加到屏体电容上，使屏体电容量增加。

在触摸检测时，自电容屏依次分别检测横向与纵向电极阵列，根据触摸前后电容的变化，分别确定横向坐标和纵向坐标，然后组合成平面的触摸坐标。自电容的扫描方式相当于把触摸屏上的触摸点分别投影到 X 轴和 Y 轴方向，然后分别在 X 轴和 Y 轴方向计算出坐标，最后组合成触摸点的坐标。见图 6-38。

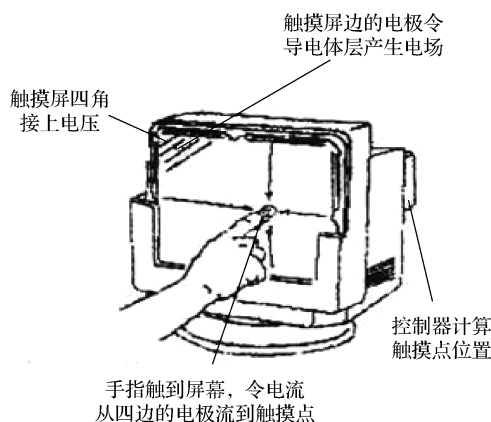


图 6-37 表面电容式触摸屏

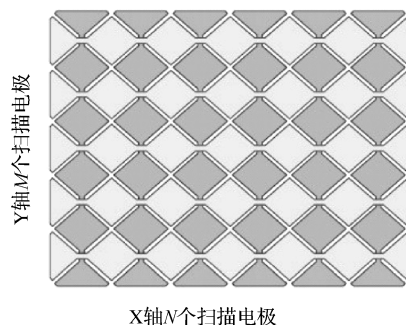


图 6-38 投射电容屏

自容式扫描的优势是扫描速度快，一个扫描周期只需要扫描 $N+M$ 条电极（ N 和 M 分别是 X 轴和 Y 轴的扫描电极数量）。如果是单点触摸，则在 X 轴和 Y 轴方向的投影都是唯一的，组合出的坐标也是唯一的；如果在触摸屏上有两点触摸并且这两点不在同一 X 方向或者同一 Y 方向，则在 X 和 Y 方向分别有两个投影，则组合出 4 个坐标。显然，只有两个坐标是真实的，另外两个就是俗称的“鬼点”。因此，自电容屏无法实现真正的多点触摸。见图 6-39。

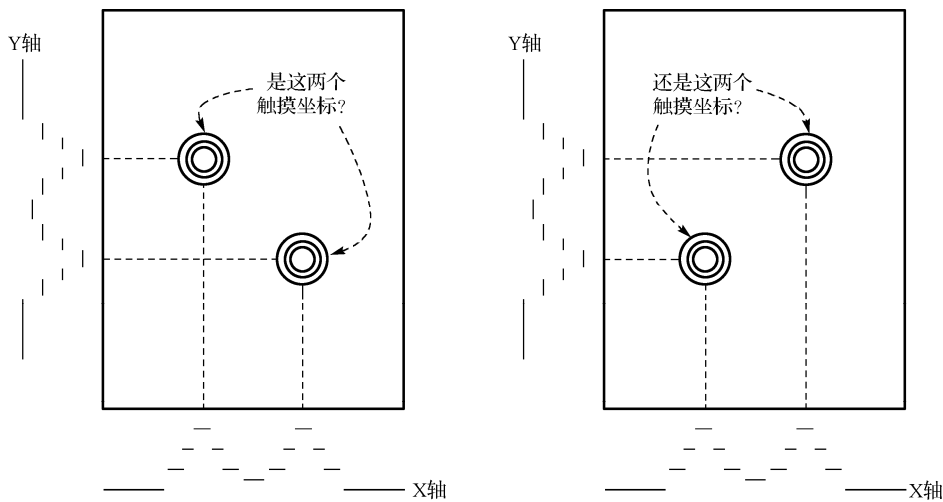


图 6-39 自容式扫描

多点触摸，顾名思义就是识别到两个或以上手指的触摸。多点触摸技术目前有两种：多点触摸识别手势（Multi-Touch Gesture）和多点触摸识别位置（Multi-Touch All-Point）。

多点触摸识别手势，即使用两个手指触摸屏幕时，也可以识别到这两个手指的运动方向，但还不能判断出具体位置，可以进行缩放、平移、旋转等操作。由于自电容触摸屏存在“鬼点”问题，只能进行多点触摸识别手势，无法实现真正多点触摸。

多点触摸识别位置可以识别到触摸点的具体位置，即没有“鬼点”的现象。多点触摸识别位置可以应用于任何触摸手势的检测，甚至可以检测到双手十个手指的同时触摸。互电容触摸屏支持多点触摸识别位置。

互电容触摸屏也是在玻璃表面用 ITO 制作横向电极与纵向电极，它与自电容屏的区别在于，两组电极交叉的地方将会形成电容，即这两组电极分别构成了电容的两极，如图 6-40 所示。当手指触摸到电容屏时，影响了触摸点附近两个电极之间的耦合，从而改变了这两个电极之间的电容量。检测互电容大小时，横向的电极依次发出激励信号，纵向的所有电极同时接收信号，这样可以得到所有横向和纵向电极交汇点的电容值大小，即整个触摸屏的二维平面的电容大小。根据触摸屏二维电容变化量数据，可以计算出每一个触摸点的坐标。因此，屏上即使有多个触摸点，也能计算出每个触摸点的真实坐标。

互电容触摸屏需要单独扫描检测行列的所有交叉点，扫描次数是行数和列数的乘积。例如，一个 10 根行线、15 根列线所构成的触摸屏，自电容式需要扫描的次数为 25 次，而互电容式则需要扫描 150 次。自电容触摸屏检测的是每条电极寄生电容的变化，有手指存在时寄生电容会增加，从而判断有触摸存在，而互电容是检测行列电极交叉处的互电容（也就是耦合电容）的变化，如图 6-40 所示。有手指存在时互电容会减小，就可以判断触摸存在，并且准确判断每一个触摸点位置。

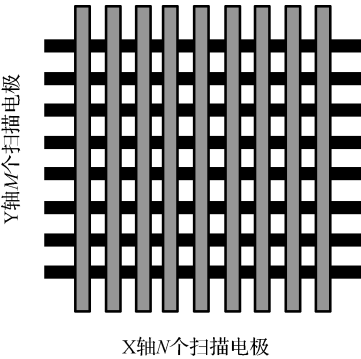


图 6-40 互电容触摸屏

2. SEP4020 微处理器的触摸屏驱动

SPI 接口用于在 SEP4020 中进行触摸屏数据采集，触摸点的数字值通过 SPI 接口送给 CPU。SPI 和触摸屏的协同工作如下所述。

先是从 SPI 中向 A/D 芯片发出 X 命令，之后的第二个时钟周期 SPI 开始接收从 A/D 芯片过来的数据，也就是 X 坐标轴的高 8 位数据，接着是第三个时钟周期，接收的是 x 坐标轴的低 8 位数据，此时 X 的模数转换结束。以同样的方式，SPI 在第三个时钟周期向 A/D 芯片发出 y 命令，之后的第四个时钟周期接收 Y 坐标轴的高 8 位数据，最后在第五个时钟周期接收到 Y 轴的低 8 位坐标值。

一般而言，触摸屏中断服务程序的主要任务是负责发命令和接收数据。首先通过中断发现笔落下，然后在中断处理程序中启动定时器（并结束 AD 中断服务程序），定时器中断服务程序会定时采样 A/D 值，直到发现没有数据时，判断为笔抬起，关闭定时器中断，完成一次完整的触摸过程。

6.4 定 时 器

6.4.1 通用定时器

硬件定时器虽然简单，但往往是计算机系统中非常重要的一个硬件模块。操作系统需要定时器产生系统内部的“嘀嗒”时钟，操作系统专用的定时器一般称为 OS tick。对于采用时间片轮转调度策略的系统而言，OS tick 的中断服务函数将触发操作系统调度器运行，以决定是否要暂停当前进程，执行下一个有效的等待任务；许多硬件驱动都需要定时器产生相应的时序（比如马达控制）等。硬件定时器的核心为硬件计数器，一般而言，计数器从预置的值开始倒数，倒数到零可以触发事件，诸如产生中断。计数器倒数到零时，通常可以自动载入预置值，重新开始倒数。

图 6-42 为通用定时器的逻辑框图。

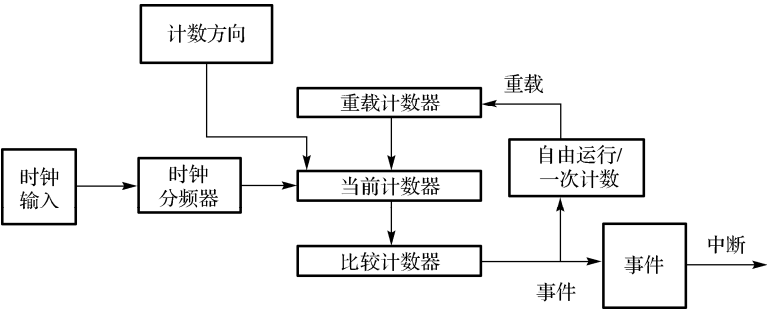


图 6-42 通用定时器结构框图

定时器的设计与使用需要注意以下参数。

- (1) 时钟源：时钟源表明计数器计数的时间间隔。定时器的计数时钟通常由系统时钟硬件分频得到，分频比通常可配置。
- (2) 计数精度：定时器的精度很大程度上由定时器的时钟源决定。时钟源的精度单位为百万分之一（PPM，Parts Per Million）。使用振荡器（通常是锁相环电路）产生的时钟源精度通常高于使用晶振产生的时钟源。在嵌入式系统中，OS tick 有时也用于标明实际时间。在这种情况下，时钟源精度则非常重要。计数一天的情况下，100PPM 的晶振可能将产生 8.64s 的误差，而 12PPM 的振荡器只产生 1s 左右的误差。需要注意的是，时钟源精度会随温度变化在标定的范围内变动。
- (3) 计数模式：定时器通常有 3 种计数模式：重启计数模式、自由运行模式（free run）与一次计数模式（one shot）。重启计数模式的定时器倒数到零后将自动重载预置的初始计数值，并重新开始计数。当计数器值为零时可以触发事件，诸如产生中断。自由运行模式在定时器倒数到零后产生中断，并自动重置计数器的值到最大值（对于 32 位定时器为 0xFFFF_FFFF，16 位定时器为 0xFFFF）重新开

始计数。一次计数模式的定时器在计数器值为零时将停止工作，定时器需要软件显式地重启计数。自由运行模式又称为周期模式，一次计算模式则为非周期模式。

(4) 计数方向：定时器的计数器可以递减计数，也可以递增计数。对于倒数的定时器，计数器为零时触发事件；对于正数的定时器，计数器为预置值时触发事件。

(5) 计数器宽度：定时器的核心是计数器，计数器在时钟的作用下递增或是递减计数，最大计数值由计数器宽度决定。对于 32 位计数器来说，最大计数值为 `0x0 FFFF FFFF`。在一些嵌入式系统中，存在一些大于 32 位的计数器。在 32 位系统中，读取这种大于 32 位的计数器，需要进行多次读操作。此时需要注意的是，由于两次读操作之间不可避免地存在延时，前后两次读出的计数器高低位值可能并不对应，从而产生较大的误差。举例来说，某一时刻 64 位递增计数器的值为 `0x0 FFFF FFFF`，需要分两次读出。如果先读出高 32 位 0，再读出低 32 位，由于计数器仍在运行，读 32 位时，计数器变为 `0x1 0000 0000`，此时读出的低 32 位也为 0，高低位拼合后的结果为 0，与实际计数值相差极大。同理，如果先读出低 32 位值，再读出高 32 位值，最后拼合的结果可能为 `0x1 FFFF FFFF`，同样误差极大。为了解决这一问题，位宽大于读数据总线的计数器通常设计有专门的读取逻辑，可以使用命令在读取前手动锁存计数器值，再读取锁存之后的计数器值；也可以在读计数器的低位（或高位）时，自动锁存高位（或低位）值，简化操作，同时减少寄存器的使用。

除了通用定时器外，嵌入式系统中一般设计有一些特殊功能的定时器。

6.4.2 RTC

前面提到，在嵌入式系统中，OS tick 有时也用于标明实际时间。在大多数嵌入式系统中，实际时间由专门的实时时钟（RTC，Real-Time Clock）产生。通过使用备用电池，系统掉电的情况下，RTC 仍能够持续工作，保持计时。

在一些需要深睡眠的嵌入式系统中，RTC 工作时的功耗是不可接受的。除此之外，当定时器、RTC 产生的实际时间不准确时，系统可以从外部获得较为准确的实际时间，对实际时间进行校正。网络时间协议（Network Time Protocol）为嵌入式系统提供了相应的时间获取及校正机制。此外，通过全球定位系统（GPS，Global Positioning System）提供的时间戳（time stamp）功能，也能进行高精度的时间校正。

RTC（real-time clock）实时时钟控制模块主要是提供系统实时时钟、产生连续中断以及计时的功能。在低功耗模式以及非外部系统复位的过程中，RTC 的系统时钟逻辑仍然保持正常工作，保证系统时钟不丢失。

RTC 具有如下特点：

- 提供年，月，日，时，分，秒计时。
- 设置定时中断（精确到分）。
- 提供 WatchDog 功能。
- 提供 1/256s~1s 软件可配置的连续中断。
- 提供闰年判断机制（只支持 2004 年、2008 年和 2012 年）。
- 32.768kHz 操作。

图 6-43 是其功能模块图。RTC 模块主要包括 TIMER、ALARM、WATCHDOG 和 SAMPLE 功能。TIMER 功能主要用于 RTC 计时。其可以计时年、月、日以及小时、分和秒。

ALARM 功能用于设置定时闹铃功能。当其设置的小时及分钟和 TIMER 的计时时刻相同时，其可以产生闹铃中断。

WATCHDOG 功能用于监视操作系统状态，防止操作系统死锁。使能 WATCHDOG 功能以后，需要软件周期性地复位 WATCHDOG 以避免其产生中断或者复位。

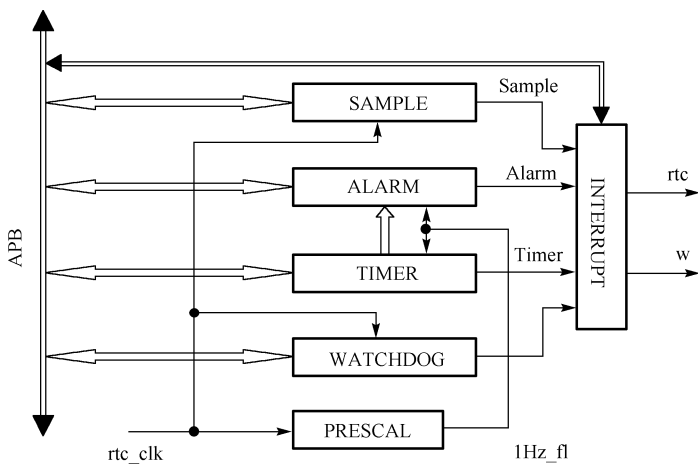


图 6-43 RTC 内部子模块框图

SAMPLE 功能可以设定一定间隔后产生中断。清除中断标志位后,经过相同的间隔其将再次产生中断。图 6-44 为使能流程,图 6-45 为关断模块流程,图 6-46 为中断服务流程。

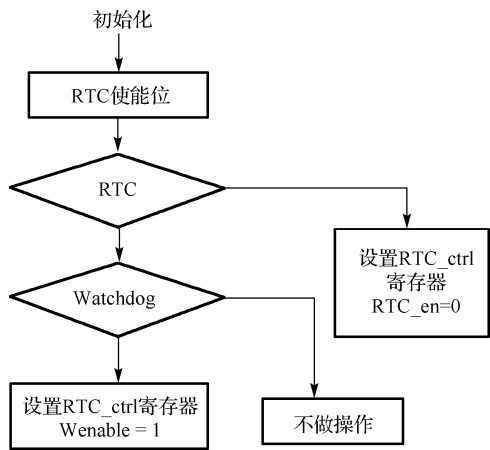


图 6-44 使能流程

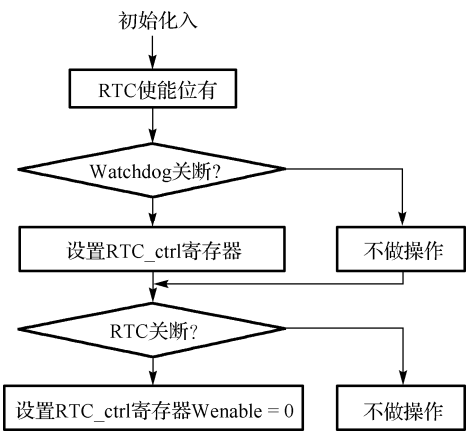


图 6-45 关断模块流程

1. Watchdog

看门狗（WDT，WatchDog Timer）是一类专用定时器，其基本组成与一般定时器类似，不同之处在于，看门狗计数完成后触发的事件不仅包括产生中断，还包括系统复位。

看门狗的作用在于，当系统由于某种原因无法正常工作时，看门狗可以及时复位系统，尝试重启系统。正常情况下，使用看门狗的系统需要定时重置看门狗计数器的值（这个过程称为“喂狗”），防止计数器计数结束。当系统由于不当操作或是设计问题陷入死循环或停止运行时，看门狗的计数器值得不到重置，将触发系统复位。

有许多方法可以保证嵌入式系统的正常运行，其中最简单的方法就是使用看门狗。使用看门狗的嵌入式系统，运行的软件需要定期访问看门狗，重置计数器，避免看门狗定时器到期。“喂狗”周期需要注意，对于重负载系统，系统可能没有时间进行过于频繁的“喂狗”操作。大多数看门狗计数器到期时间为 1~2s。对于复杂系统，可以使用操作系统的任务内容切换及来自应用任务的软件通知，检查系统是否正常运转。

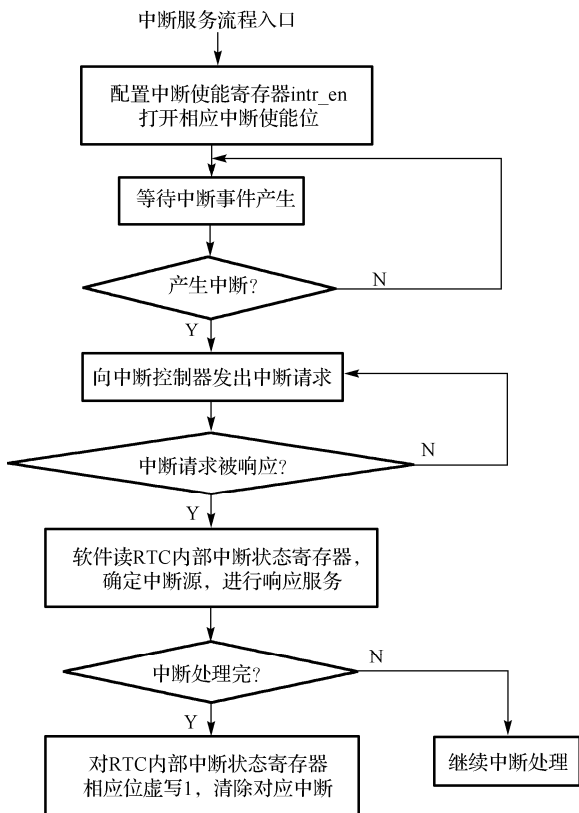


图 6-46 中断服务流程

为了防止对于看门狗的意外操作，像是错误“喂狗”，对于看门狗寄存器的操作通常需要按照一定的特殊规范进行。

SEP4020 中的 Watchdog 模块主要是提供防止系统失败的一种保证措施，通过软件间隔地对 Watchdog 进行服务，确保系统工作正常。其特征如下：

- 提供 0.5~64s 的精度为 0.5s 的间隔范围。
- 提供中断 / reset 可选输出。

当软件没有对 Watchdog 进行服务时，将会产生 timeout 事件，通过配置中断使能寄存器中的中断位和 reset 请求位可以产生中断和 reset 请求，如果需要屏蔽也可以通过配置这两位来获得（默认情况为屏蔽）。

Watchdog 计数器重新装载的触发事件有以下几个：

- 对服务寄存器进行虚写 0XAAAA（重新装载）。
- reset 复位（此时为初始值）。
- Watchdog 中断清除。

思考题

1. 请读者思考如何编写外设接口控制器的驱动程序？驱动程序是如何与 SEP4020 芯片中的外设接口控制器进行交互的？
2. 请读者思考在嵌入式系统的开发中都有哪些情况需要用到 SEP4020 芯片中的定时器模块？如何使用？

3. 请读者思考如何设计 SEP4020 中的简单功能模块——键盘接口并编写键盘接口的驱动程序？

扩展阅读

- [1] IEEE Standard for Information technology-Telecommunications and information exchange between system-Local and metropolitan area networks-Specific requirements-Part3: Carrier sense multiple access with collision detection(CSMA/CD)access method and physical layer specification[S]. IEEE Std 802. 3, 2000 Edition. 2000, 33-254.
- [2] Tang Ken. Challenges of medium access control (MAC)protocols in ad hoc mobile networks[C]. Dissertation Abstracts International. 2002, 63(7):3362.
- [3] Universal Serial Bus Specification Revision 1. 1 [S]. USB Implementers Forum. 1998.
- [4] USB2. 0 Transceiver Macrocell Interface (UTMI)Specification. Version 1. 05. Intel. 2001.
- [5] Gen Chandler, Chris Kolb, Maria Polhman, et al. On-the-go supplement to the USB2. 0 specification [EB/OL]. (2006-04-04) www. usb. org.
- [6] Universal Serial Bus Mass Storage Class Specification Overview Revision 1. 2 [S]. USB Implementers Forum. 2003.
- [7] Richard Wagner. Professional iPhone and iPod Touch Programming: Building Applications for Mobile Safari[M]. JOHN WILEY & SONS INC, 2008.
- [8] Yoshio ITAKURA. Trends in Touch Screen Technology and Material[J]. Journal of Printing Science and Technology, 2010, Vol. 47(6), pp. 387-391.
- [9] Zongging Lu. A Defect Inspection Algorithm for LCD Screen Touch [J]. The 1st International Conference on Information Science and Engineering, 2009:103-1034.
- [10] Chen, Shuangye, Qin, Xiaoyu. The research of touch-screen calibration algorithm and its application to the embedded system [J]. IEEE Conference Publications, 2012, 483-486.
- [11] Philips Semiconductors I2S bus specification. <http://www.semiconductor.philips.com> 1996.
- [12] 宋宝华. Linux 设备驱动开发详解[M]. 北京：民邮电出版社，2008. 276-285.

第7章 嵌入式系统软件概述

如果说硬件系统是一个人的身体的话,那么软件系统无疑是这个人的灵魂。说得更加专业化一点,硬件提供了系统功能的所有机制,而软件则实现了系统功能的所有策略。软件响应用户的输入,操作和控制具体的硬件,并将运算的结果最终通过硬件展现(输出)给用户。因此嵌入式系统是一个软硬件协同工作的系统,缺一不可。本章将首先介绍嵌入式系统的软件框架,包括嵌入式软件所面临的挑战以及嵌入式软件的层次框架;作为嵌入式软件系统的核心,嵌入式操作系统的作用正在变得越来越不可或缺,本章将向读者(尤其没有操作系统知识的读者)介绍嵌入式操作系统的基本原理;毫无疑问,Android 操作系统已经成为智能手机领域最重要的操作系统(几乎没有“之一”了),在本章的最后,我们将简要介绍这个操作系统的基本框架和原理。

7.1 嵌入式系统的软件框架

7.1.1 嵌入式系统软件所面临的挑战

比硬件系统按照摩尔定理的速度飞速发展更加惊人的是软件规模的急剧膨胀。在 65 纳米的工艺下,软件的研发成本已经占到整个 SoC 研发成本的 56%,而且有充分的理由相信这个比例还会随着工艺的提升进一步提高。而这仅仅是芯片设计阶段的成本,对于系统产品而言,另一个与之对应的数据是,从 1992 年到 2004 年,嵌入式硬件的规模增长了 43 倍,与此同时软件的规模却增长了令人惊讶的 900 倍!

与软件规模不断膨胀相适应的是软件开发成本的急剧上升和软件质量控制的难上加难。据一组国外研究者 2003 年给出的数据显示,54%的嵌入式软件项目开发进度落后于计划;66%的软件项目开发成本最终超出预算;33%的软件不能满足产品在功能和性能上的需求。给出这组数据的风河公司工程师在内部将这组数据称为“恐怖统计”(Statistics of Doom)。而随着越来越多的设备需要与互联网连接起来,软件开发的压力还会进一步加大。

总的来说,现代嵌入式软件所面临的挑战主要包括:第一,软件功能的急剧膨胀,越来越多的功能被集成到设备中。甚至很多传统的简单设备在升级阶段都集成了更加复杂的功能,这最主要体现在对互联性的要求上,比如随着物联网应用的兴起,几乎所有的传统设备都需要增加网络功能。而传统的通过数码管、LED 和按键作为人机交互的应用,开始采用彩色触摸屏,比如智能冰箱甚至彩屏电饭煲。另一方面,在移动互联网领域更是逐渐集成了传统 PC 的所有功能,甚至有过之而无不及。比如,实时的高清视频编解码、语音识别与交互(想想苹果的 Siri 吧)、3D 游戏等。这些新集成的功能大大增加了软件的复杂度。

第二,随着软件复杂的增加,软件开发和测试的难度越来越大。熟悉软件工程的读者都知道,软件开发和测试的难度与软件规模的关系不是简单的线性关系,而是随着规模的增加,开发与测试难度急剧增加。另外,由于功能复杂性的增加,开发者可能需要从不同的软件中间件提供商处购买不同的软件中间件,这也在很大程度上增加了集成的复杂度和难度。

第三,软硬件适配与优化对于提升系统的总体性能已经具有举足轻重的作用。随着现代高性能 SoC 集成的模块越来越多,越来越复杂,对于底层软件,尤其是操作系统的适配与优化已经成为提升系统

总体性能的关键。以 Android 系统为例，虽然 Google 发布了 Android 系统的所有源码，但为了使得该系统能够最大程度地发挥硬件的性能，几乎所有的整机厂商都必须对 Android 系统进行相关的定制优化。这还不包括 ARM 公司与 Google 合作进行的 Android 底层代码针对 ARM 架构的性能优化，比如 Thumb2 指令集优化，通过 Neon SIMD 指令集对底层 Skia 图形库的汇编优化，等等。这些底层代码的适配与优化对于系统设计人员提出了巨大的挑战，一方面需要优化人员对所采用的 SoC 芯片具有非常深入的理解，另一方面也要求他们对 Android 系统的底层实现具有非常深入的理解。

针对现代嵌入式软件开发所面临的这些挑战，工业界主要通过两个途径来解决。第一，操作系统厂商将越来越多的功能集成到操作系统中，使得整机设计人员，尤其是应用开发人员能够将主要的工作精力集中在应用程序本身，而将硬件的复杂性通过应用程序接口屏蔽掉。早期的嵌入式操作系统往往只是提供简单的内核功能，只负责管理有限的任务管理、任务间通信、中断管理和内存管理等最基本的功能。而最新的嵌入式操作系统已经集成了不亚于桌面级操作系统的功能，从网络协议栈、文件系统、网络浏览器、数据库、图形用户界面（GUI）到 Java 虚拟机等。有些操作系统，比如 Android 甚至集成了部分常用的应用程序，比如短信、电话、通信簿等。这些被集成的软件模块大大减轻了应用设计人员的工作量。第二，涌现出越来越多的设计工具来帮助系统设计人员优化整个系统的性能。早期的设计工具只包括工具链（编译器、汇编器、连接器和标准 C 库），而现代的嵌入式操作系统厂商不仅仅提供图形化的集成开发工具将基于命令行的工具链整合（通常还会包括一个基于 PC 的操作系统模拟器），甚至还提供包括线程级调试的分析工具，帮助开发人员跟踪、分析整个系统的性能瓶颈。

7.1.2 嵌入式软件的层次框架

正如前一节所介绍的，为了适应应用不断复杂化的需求，现代嵌入式操作系统已经演化为一个非常复杂的软件系统。工程中实现一个复杂系统的通常做法是分而治之，将一个复杂系统划分为若干个层次，每个层次再进一步划分成为若干个模块。每个模块实现一个相对简单的功能，并通过定义的相应接口与其他模块进行交互；若干模块共同组成一个功能层次，层次之间也通过相对固定的接口与上层和下层进行交互，从而实现整个系统的完整功能。

现代嵌入式软件系统也是按照这样的层次和模块系统进行划分的，不同系统的具体划分方法以及相应的接口定义都各不相同，然而我们依然能够对一个嵌入式软件系统进行粗略的划分，如图 7-1 所示。总的来说，一个嵌入式软件层次可以从上到下（我们把应用程序作为整个层次结构的最上层，而把硬件系统作为最下层）划分为应用层、应用接口层、中间件、操作系统层、硬件驱动层、板级支持包和硬件层。通常情况下，应用层是和用户交互的层面，实现嵌入式系统的具体功能，比如洗衣机中的洗衣模式控制程序，再比如智能手机中的 3D 游戏和电话程序等。

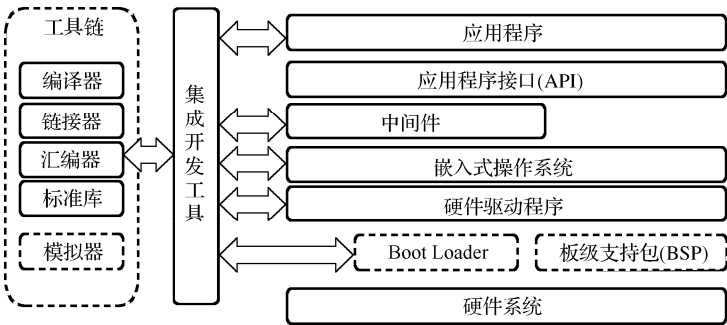


图 7-1 嵌入式软件系统的层次框架

应用程序的编写者通常并不需要知道具体硬件平台和操作系统的实现细节，应用程序员通过操作系统提供的编程接口（有些时候还包括中间件所提供的编程接口）访问系统所提供的各项服务。这些编程接口主要体现为一组预先定义的函数和相应的数据结构，对于应用程序员而言它们就像是黑盒一样，只需要输入正确的参数就可以获得正确的结果或者相应的硬件操作，而无需知道细节。它们通常被称为应用程序编程接口（API, Application Programming Interface）。不同的操作系统会提供不同的 API 接口，也就是不同的函数接口，这为应用程序在不同操作系统之间的移植带来了困难，因此主流的操作系统一般都会实现遵循某些标准的 API 接口以方便应用程序的移植工作，比如 Linux 操作系统所支持的 POSIX 标准和 Pthread 标准等。

所谓中间件，顾名思义是介于操作系统和应用程序之间的软件层次。通常它们是由一些专业厂商提供的专业软件模块，这些软件模块实现一些特定的功能，这些功能往往是操作系统中未曾实现或者实现效果并不是太好的模块。比如，语音识别软件，语音合成软件（TTS, Text to Speech），嵌入式数据库，指纹识别软件，等等。这些软件模块通过实现相应操作系统的接口从而能够方便地集成到这些操作系统中，并对应用层提供与操作系统兼容的 API。通常中间件提供商所提供的软件模块都包含这些公司的核心技术和知识产权，因此所提供的中间件一般都以二进制库的形式发布。

嵌入式操作系统是嵌入式软件系统的核心，它对上屏蔽了硬件的所有细节，使得应用程序通过 API 可以方便地访问硬件所提供的功能；另一方面，操作系统还提供了多任务的环境，对用户创建的任务进行管理和调度，使得应用程序可以方便地实现多任务。针对应用环境的不同，系统开发人员可以选择不同的嵌入式操作系统，它们的功能和性能指标各不相同，从简单的 uCOS/II 到复杂性不亚于桌面操作系统的 Android。我们将在下一节比较详细地介绍嵌入式操作系统的基本概念和原理。

理论上来说，硬件驱动层应该是属于操作系统层的一部分，因为几乎所有的操作系统都严格地定义了硬件驱动的基本接口和操作流程。也就是说硬件驱动层一定是和相应的操作系统紧密绑定的，很难存在一个独立于操作系统的硬件驱动。但由于驱动层对于整个软件系统的重要性，我们还是将它作为一个单独的层次进行介绍。操作系统通过驱动层的软件具体实现对于硬件设备的操作和管理，系统对于所有硬件（CPU 本身除外）的具体操作都是通过驱动软件实现的。因此，驱动层实际上是对真实硬件的抽象与屏蔽，使得上层的操作系统可以通过统一的方式对硬件进行管理。

在硬件驱动之下应该就是硬件层了，但是在图中我们却用虚线框增加了 Boot Loader 和板级支持包（BSP）两个软件模块，因为这两个软件模块不管是在功能上还是在名称上经常会给初学者造成困惑。首先介绍 Boot Loader，顾名思义这是一个用于系统启动的软件模块。通常它既不属于驱动软件，也不属于操作系统，其主要功能是系统上电时首先接管整个系统并完成必要的系统自检和部分关键硬件模块的初始化。比如，初始化锁相环以使芯片工作在正常频率，初始化 DDR 控制器以使得 CPU 可以访问 DDR 存储器，等等。在完成硬件的必要初始化后，Boot Loader 将从指定的非易失存储器（可以是 Flash 存储器，也可以是 SD 卡等外置存储器。当然在使用这些外设之前，Boot Loader 也必须完成对它们的初始化并加载相应的驱动）中读取操作系统的镜像，并将其装载到相应的存储器地址（通常在 DDR 存储器中），在完成了这个装载后，Boot Loader 会将控制权交给装载完成的操作系统，最终实现操作系统的启动。板级支持包是由开发板厂商或者芯片厂商在发布相应的评估板时一起发布的用于支持开发板基本硬件的底层软件。BSP 一般独立于操作系统，也就是 BSP 不依赖于任何操作系统（不调用任何操作系统提供的系统调用），开发人员可以基于 BSP 实现对该开发板的基本测试和使用，他们甚至可以基于 BSP 所提供的源码实现相关操作系统的硬件驱动软件。从某种角度上说，BSP 的功能与 PC 上的 BIOS 相类似，它提供了对开发板上硬件的最基本的操作。

与嵌入式软件系统相对应的是嵌入式软件开发工具。正如上一节介绍的，早期的开发工具基本上只包括最基本的工具链（编译器、汇编器、链接器和标准库），但是随着嵌入式软件开发难度的不断增

加，现代的嵌入式操作系统提供商通常会提供集成了工具链、操作系统模拟器、软件开发项目管理器、甚至更多高级特性的图形化集成开发工具软件。这些高级特性可能包括对用户开发软件进行线程级的分析或者是对 SoC 中集成的性能监测单元（PMU，Performance Monitor Unit）中的实时性能参数的读取等，关于开发工具的内容可以参阅第 3 章。

7.2 嵌入式操作系统的基本原理

7.2.1 嵌入式操作系统简介

为什么要使用嵌入式操作系统呢？早期的嵌入式系统开发一般都是由一个工程师完成的，软件开发工作只占全部工作的 5%~10%；随着科技的发展，20 世纪 80 年代软件开发工作已经占到全部工作的 50%；近几年，随着硬件复杂性、多样性和应用复杂性的增加，软件开发工作急剧增长，经常达到全部工作的 70%~80%。传统的开发模式已经不能适应系统复杂性的增长，而嵌入式操作系统的引入极大地方便了嵌入式软件的开发和维护。

嵌入式系统开发平台的引入，极大地方便了嵌入式软件的开发和维护。嵌入式操作系统体现了一种新的系统设计思想和一个开放的软件框架，软件工程师只做少量改动，就可以添加或删除一个系统模块。通过操作系统所提供的应用程序编程接口（API）访问系统资源，使得应用软件工程师能够将精力集中于所要解决的问题，而不是烦琐的系统底层操作，提高了开发效率。它解决了嵌入式软件开发标准化的问题，更好地支持了系统协同开发。基于嵌入式操作系统开发出的程序具有较高的可移植性，能实现 90%以上的设备独立。

嵌入式操作系统的特点：

- 高效的任务管理。
- 支持多任务；优先级管理；任务调度：基于优先级的抢占式调度、时间片轮转调度的算法；支持快速而确定的上下文切换。
- 快速灵活的任务间通信：信号量、消息队列、管道等。
- 高度的可剪裁性。
- 动态链接与部件增量加载。
- 快速有效的中断和异常事件处理。
- 优化的浮点支持。
- 动态内存管理。
- 系统时钟和定时器。

下面是几种常见的 RTOS：

- 软实时 RTOS：嵌入式 Linux；Android。
- 硬实时 RTOS：VxWorks；Nuclear。
- 著名的 open RTOS：ucOS/II RTEMS。

嵌入式操作系统（RTOS 或 EOS，real-time embedded operating system）是一种实时的、支持嵌入式系统应用的操作系统软件，它是嵌入式系统（包括硬、软件系统）极为重要的组成部分，通常包括操作系统内核、与硬件相关的底层驱动软件、设备驱动接口、通信协议、图形用户界面（GUI）、网络浏览器等。

与通用操作系统相比，嵌入式操作系统在系统实时高效性、硬件的相关依赖性、软件固态化以及应用的专用性等方面具有较为突出的特点。同时，嵌入式操作系统可以提供高效的任務管理，包括对

多任务的支持、优先级管理和任务调度，并支持快速的上下文切换，嵌入式操作系统可以提供快速灵活的任务间通信，任务间通信的方式有信号量、消息队列和管道等。

另外，嵌入式操作系统一般具有高度的可剪裁性，可以实现动态链接与部件增量加载，还可以进行快速有效的中断和异常事件处理，并提供优化的浮点支持、动态内存管理、系统时钟和定时器，等等。

实时多任务系统主要完成任务切换、任务调度、任务间通信、同步、互斥、实时时钟管理、中断管理等。同时实时操作系统应具备异步的事件响应、切换时间和中断延迟时间确定、优先级中断和调度、内存锁定技术、连续文件技术、提供任务间同步、协调各个任务对共享数据的使用。

系统设计人员可选择的嵌入式操作系统有很多，一般来说可以将嵌入式操作系统分为软实时系统和硬实时系统。软实时系统的宗旨是使各个任务运行得越快越好。硬实时系统各个任务不仅要执行无误而且要做到实时。较为流行的嵌入式操作系统有：嵌入式 Linux、Andorid、iOS、Win CE、Sybian 和 Palm OS 等，这些为软实时的 RTOS；硬实时 RTOS 有 VxWorks、Nuclear、pSOS 和 OS-9；著名的 open RTOS 有 uCOS/II RTEMS 等。

随着移动互联网应用的兴起，ARM 架构已经成为了移动互联网终端（比如手机和平板电脑）的主流处理器架构，而在应用于此类终端产品的嵌入式操作系统则越来越向 Android 和 iOS 两大阵营集中。在工业控制及其他对实时性要求较高的领域，VxWorks 作为老牌的强实时操作系统一直居于不可撼动的领先地位，但随着处理器的计算性能越来越强大，嵌入式 Linux 因其开源和免费的优势，越来越受到工程师们的欢迎。在低端应用市场，随着 32 位微控制器（MCU）的普及（以 ARM 公司的 Cortex M 系列为代表），uCOS 操作系统也得到了比较广泛的应用。

嵌入式 Linux 操作系统的迅速崛起，主要由于人们对自由软件的渴望与嵌入式系统应用的特殊性，要求提供系统源码层次上的支持，而嵌入式 Linux 正适应了这一需求，它不仅具有开放源代码，而且系统内核小、效率高、网络协议完整，裁减后的系统很适于如信息家电等嵌入式系统的开发。

随着半导体行业的迅猛发展以及移动互联网时代的来临，各类移动通信设备正在以前所未有的速度快速普及。以往各大手机厂商采用的封闭式操作系统也被各种各样的智能手机操作系统取代。人们对于手机功能的使用，也已不再局限于最基本的通话功能了，手机已经与人们的生活紧密地结合在了一起。由 Google 公司领导及开发的智能手机操作系统 Android 就是其中之一。Android 平台由于具有开源许可、定制性强等特点，发展非常迅猛。2011 年第一季度，Android 在全球的市场份额首次超过塞班系统，跃居全球第一。2013 年的第四季度，Android 平台手机的全球市场份额已经达到 78.1%，其在中国市场占有率更是高达 90%。Android 已经渗透到人们工作生活的方方面面。

VxWorks、pSOS 和 OS-9 是传统嵌入式操作系统领域中应用最广泛、市场占有率较具优势的几个系统。它们是专门为嵌入式微处理器设计的高模块化、高性能的实时操作系统，广泛应用于高科技产品中，包括消费电子设备、工业自动化、无线通信产品、医疗仪器、数字电视与多媒体设备，具有很好的安全性、容错性及系统灵活性。

7.2.2 嵌入式操作系统的内核

在了解嵌入式操作系统的内核结构之前，我们先来了解两个概念：单内核与微内核。单内核结构是在硬件之上定义了一个高阶的抽象界面，应用一组原语（或者叫系统调用）来实现操作系统的功能，例如进程管理、文件系统和存储管理等，这些功能由多个运行在核心态的模块来完成。单内核（monolithic kernel）是一个很大的进程，它的内部又可以被分为若干模块（或者是层次）。但是在运行的时候，它是一个独立的二进制镜像。其模块间的通信是通过直接调用其他模块中的函数实现的，而不是消息传递，如图 7-2 所示。

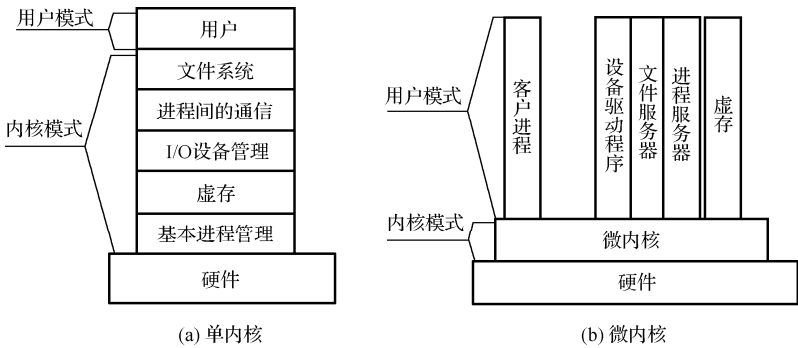


图 7-2 内核架构模式

微内核（microkernel）最根本的思想是要保持内核尽量小，这样只需要把微内核本身进行移植就可以将整个内核移植到新的平台上。它的基本原理是只把操作系统最基本的功能才放在内核中，其他模块都作为系统服务（通常是一组进程或线程）运行于微内核之上，并不直接依赖硬件。微内核结构用一个水平分层结构代替了传统（单内核）的纵向分层结构。微内核操作系统的大部分功能都作为独立的进程（也就是系统服务，Services）在特权状态下运行，它们通过消息传递进行通信。在典型情况下，每个功能模块都有一个进程。因此，如果在设计中有一个系统调用模块，那么就必然有一个相应的进程来接收系统调用，并和能够执行系统调用的其他进程（或模块）通信以完成所需任务。微内核架构的一个优点是在不影响系统其他部分的情况下，可以方便地增加或卸载相应的服务模块，因为每个服务模块都是作为独立的系统服务存在的。我们甚至可以在系统运行时将开发出的新系统模块加入系统。另外一个优点是不需要的模块将不会被加载到内存中，因此，微内核就可以更有效地利用内存。另一方面，由于用户程序访问相应的系统功能实际上最终需要依靠进程间通信机制来实现，也就是我们常说的客户端-服务器（Client-Server）模型，在效率上微核架构的操作系统可能要比单核系统低。

正如 CPU 架构被分为 RISC 与 CISC，但现代微处理器其实很难被称为纯粹的 RISC 或 CISC 一样，现代嵌入式操作系统也很难被简单地分为纯粹的单内核与微内核架构。一般而言，Linux 被称为单核架构，但即使这样，Linux 也引入了所谓内核线程的概念，通过这些线程实现一些内核功能。而对于架构在 Linux 之上的 Android 操作系统而言，其核心理念却比较接近微内核的设计思想。所有的 Android 功能基本上都是通过一系列的 Service 线程来实现的（为此 Android 中甚至引入了专门管理这些服务的 Service Manager，并通过 Binder 机制实现几乎所有的服务访问）。无论如何，由于微内核架构的优势，同时也由于嵌入式微处理器性能越来越强，操作系统中越来越多地引入了微内核架构的设计思想。

微核架构的设计思想是内核只实现操作系统最基本的功能，包括任务管理（创建、调度等）、任务间通信、中断管理等。我们将在本节后面分别介绍它们。

7.2.3 任务管理与调度

1. 任务与多任务

多任务运行的实现实际上是靠 CPU 在许多任务之间转换、调度。对于单处理器的系统而言，CPU 只有一个，轮番服务于一系列任务中的某一个。多任务使 CPU 的利用率得到最大的发挥，并使应用程序模块化。在实时应用中，多任务化的最大特点是，开发人员可以将很复杂的应用程序层次化。使用多任务，应用程序将更容易设计与维护。

一般而言，操作系统通过一个称为 TCB（Task Control Block）或 PCB（Process Control Block）的数据结构来管理所有的任务。每个被创建的任务占据一项 TCB，所有的 TCB 之间通过一个双向链表链接在一起，便于操作系统管理。TCB 中一般保存了与该任务相关的各项信息（不同的操作系统可能不同），比如该任务的名字、ID 号（用于 OS 内部唯一标识该任务）、该任务的入口地址（也就是该任务的入口函数指针）、任务的优先级、该任务所使用的文件句柄等。在 TCB 中有两项非常重要的信息，一是该任务的堆栈指针，其次是该任务的上下文（Context，需要说明的是不同的操作系统保存上下文的方法可能不同，某些操作系统是将上下文作为 TCB 的成员进行保存，而有些操作系统则是在内存中另外开辟一块存储区保存该任务的上下文，TCB 中只保存指向该上下文的指针）。

从代码角度看，任务（TASK）就是一个拥有自己堆栈的函数调用序列。程序的堆栈中保存了所有的函数调用顺序，程序通过构建的栈帧保存函数的返回地址、被调函数的临时变量等重要信息。每个任务作为独立的执行流，如果通过自己的堆栈保存了该执行流的所有函数调用顺序，同时再通过上下文保存了当前时刻的所有 CPU 寄存器的值，就可以切换到其他的任务执行流（被切换过来的任务也拥有自己的堆栈，其中保存了该任务的函数调用顺序，同时该任务的上下文中保存了其上次被切换前的最后 CPU 状态。操作系统只需要将上下文的内容恢复到 CPU 中，并将堆栈指针寄存器的内容恢复到该任务的堆栈指针即可完成任务的切换），如图 7-3 所示。

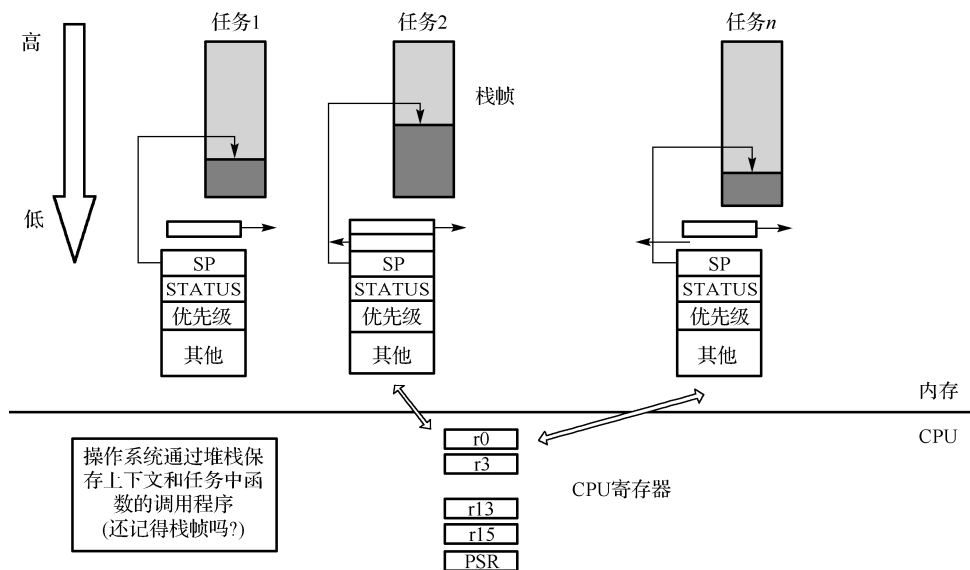


图 7-3 任务 TCB 以及任务栈的示意图

Context Switch 在有的书中翻译成上下文切换，实际含义是任务切换或 CPU 寄存器内容切换。当多任务内核决定运行另外的任务时，它保存正在运行任务的当前状态（Context），即 CPU 寄存器中的全部内容。保存工作完成以后，就把下一个将要运行的任务的上下文状态从该任务的数据结构中重新装入 CPU 的寄存器，这样这个任务将从上次被打断的地方恢复运行。这个过程叫作任务切换。任务切换过程增加了应用程序的额外负荷。CPU 的内部寄存器越多，开销就越大。进行任务切换所需要的时间很大程度上取决于 CPU 有多少寄存器需要保存和恢复。

2. 任务、进程与线程

很多没有操作系统基础的读者会被任务（Task）、进程（Process）和线程（Thread）的名字搞得莫名其妙，本小节将简单介绍它们的区别。任务通常就是线程，下面主要阐述进程与线程的区别。

一个进程拥有自己的独立内存空间，进程间的内存空间彼此隔离，以此实现保护。所以进程的实现有赖于硬件的支持（通常是内存管理单元 MMU，本书第 5 章 5.3 节介绍过虚拟内存的内容）。早期的操作系统通常都是通过进程来实现应用程序之间的隔离与保护的，每个进程中只有一个代码执行流，也就是说在这个进程构建的内存空间中只有一个程序在执行。操作系统内核负责调度不同的进程在 CPU 上运行。由于每次进程的切换都意味着内存空间的重新映射和其他程序运行环境的切换，对于这样的系统而言，实现多任务的代价是非常大的（因为每个进程中只有一个执行流）。

为了解决这个问题，现代操作系统引入了线程的概念。所谓线程就是指一个独立的程序执行流，每个线程都拥有自己独立的堆栈，是最小的可调度单元。若干个线程可以运行在一个内存空间中，这时我们称其为运行在同一个进程（空间）中。如果操作系统中同时拥有多个不同的内存空间（也就是多个不同的进程），每个进程中都可以运行各自的线程。处于同一个内存空间中的线程发生调度时不需要进行内存空间的切换，线程间的通信也要简单得多，因此较之以前每个执行流都拥有自己的内存空间的情况要高效得多。这时的进程概念更像是一个容器的概念，这个容器规定了其内部的线程所拥有的内存空间。由于线程概念的引入，通常我们将之前的单执行流进程系统称为多进程单线程系统，而对于每个进程空间中可以运行多个执行流的系统称为多进程多线程系统，如图 7-4 所示。

对于许多嵌入式系统而言，由于很多嵌入式微处理器没有 MMU，所以每个可调度单元虽然拥有自己的堆栈，但都运行在同一内存空间，我们称每个执行流为一个任务，也是一个线程。由于这些系统中的所有线程都运行在同一个内存空间，因此有时也称其为单进程多线程系统，如图 7-4 所示。

3. 任务的状态

操作系统为每个创建的任务赋予一定的状态，不同操作系统中所实现的状态可能各不相同，但最基本的状态包括运行态（Run）、就绪态（Ready）和等待态（Wait）。不同的操作系统可能扩展其他的状态，比如休眠态（Dormant）、就绪态（Ready）、运行态（Run）、等待态（Wait）、挂起态（Suspend）和等待挂起态（Waitsuspend），如图 7-5 所示。休眠态相当于该任务驻留在内存中，但并不被多任务内核所调度。就绪态意味着该任务已经准备好，可以运行了，但由于该任务的优先级比正在运行的任务的优先级低，还暂时不能运行。运行态的任务是指该任务掌握了 CPU 的控制权，正在运行中，对于单 CPU 的系统，有且仅有一个任务处于运行态。等待态指该任务在等待某一事件的发生，例如等待某外设的 I/O 操作、等待某共享资源、等待定时脉冲的到来或等待超时信号的到来等。挂起态是处于就绪态的任务被其他任务所挂起，等待挂起态是处于等待态的任务被其他任务所挂起。被挂起的任务暂时不参与系统的调度，直到其被其他任务执行恢复操作（Resume）。

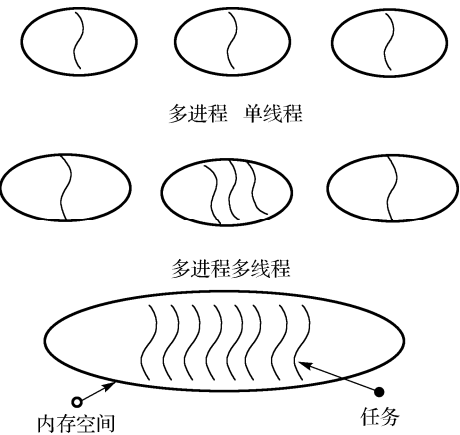


图 7-4 进程、线程、任务间的关系

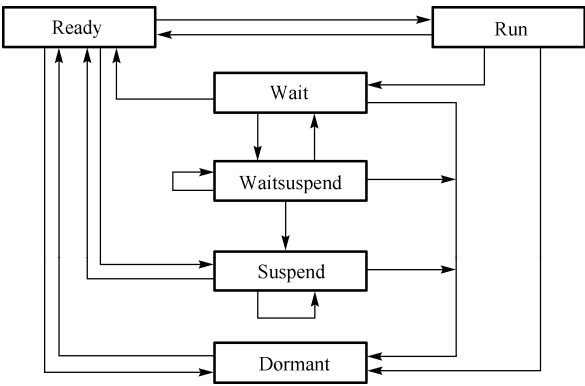


图 7-5 任务的状态

Ready、Wait 和 Run 这 3 个状态是一个多任务系统中必不可少的，其他状态因各个 OS 的不同而不同。

4. 任务的调度

在多任务系统中，每个任务要么正在使用 CPU (Run 状态)，要么正在等待 I/O 的执行或某个事件的发生 (Wait 状态)；或者该任务一切就绪，只等待 CPU (Ready 状态)。实现多任务的关键是调度器 (Scheduler)，英文还有一词叫 Dispatcher，也是调度器 (或分发器) 的意思。调度是内核的主要职责之一，也就是决定该轮到哪个任务运行了。早期的操作系统研究重点关注的是如何保证调度的公平性和效率，因此开发出了各种各样的调度算法，基本的调度算法有先来先服务 FCFS、最短周期优先 SBF、优先级法 (Priority)、轮转法 (Round-Robin)、多级队列法 (multi-level queues)、多级反馈队列 (multi-level feedback queues) 等。

与传统的桌面操作系统或服务器操作系统不同，嵌入式系统往往强调效率高于公平（这一点在面向实时应用的场合更是如此），因此多数实时内核是基于优先级调度的多种方法的复合。基于优先级的调度算法是指：每个任务根据其重要程度的不同被赋予一定的优先级，操作系统总是让处在就绪态的优先级最高的任务先运行。然而，究竟何时让高优先级任务掌握 CPU 的使用权又有两种不同的情况，所谓不可剥夺型内核（非占先式）还是可剥夺型内核（占先式）。

在不可剥夺型内核中，若一个低优先级的任务正在处于运行状态，而此时的一个中断事件（或者由于该低优先级任务调用的一个系统调用）激活了一个处于等待状态的高优先级任务，使其由等待状态进入就绪状态，但操作系统并不会立即调度该高优先级任务进入运行态，而是必须等待正在运行的低优先级任务主动放弃 CPU（比如该低优先级任务需要等待一个事件发生），操作系统才会将已经处于就绪态的高优先级任务调度给 CPU（见图 7-6）。

这也是为什么该类型的操作系统也被称为协作式多任务的原因（早期的 Windows 3.0 系统采用的就是这样的多任务机制，直到 Windows 95 才采用可剥夺型内核）。不可剥夺型内核的优点包括：第一，由于任务在运行过程中不用担心被其他任务打断（中断依然可以发生），因此在任务的代码中可以安全使用不可重入函数（前提是中断处理程序不会调用这些不可重入函数，并且该不可重入型函数本身不会调用

放弃 CPU 控制权的系统调用）。关于函数重入的问题，我们会在 7.5.2 节讨论。第二，不可剥夺型内核中几乎不需要使用信号量保护共享数据。运行着的任务占有 CPU，而不必担心被别的任务抢占。但这也不是绝对的，处理共享 I/O 设备时仍需要使用互斥型信号量。例如，在打印机的使用上，仍需要满足互斥条件。第三，由于在中断处理程序中不会发生调度，对于不可剥夺型内核而言只会在可能改变任务状态的系统调用中才会调用调度器进行调度，因此该类型操作系统的实现相对比较简单。

与之相反的是可剥夺型内核，一旦中断或一个系统调用激活了更高优先级的任务，操作系统就会立即将 CPU 调度给该高优先级任务，而强制将正在运行的低优先级任务由运行态切换到就绪态（因为这个任务是被强行剥夺的，它不等待任何其他事件，除了 CPU），关于可剥夺内核的原理见图 7-7。使用可剥夺型内核时，应用程序不应直接调用不可重入型函数。这是因为调用不可重入函数时，如果低优先级的任务在被调度前恰好运行在该函数中，而被激活的高优先级任务再次调用该函数时，会造成该函数的重入并破坏低优先级任务在该函数中的数据。如果一定要使用不可重入型函数，在调用该函数前首先应该采用信号量进行加锁，以防止其他试图调用该函数的高优先级任务暂时不能抢占该任务，

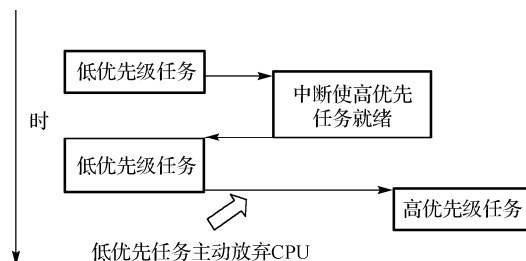


图 7-6 不可剥夺型内核的调度

在离开该函数后，应该调用信号量系统调用进行解锁。显然，可剥夺型内核的实时性更好，因为一旦高优先级任务就绪，低优先级任务必须退出执行（除非发生优先级反转，我们将在下一小节介绍这个问题），从而保证了高优先级任务的实时性。因此基本上所有的面向实时应用的嵌入式操作系统都需要支持可剥夺内核。

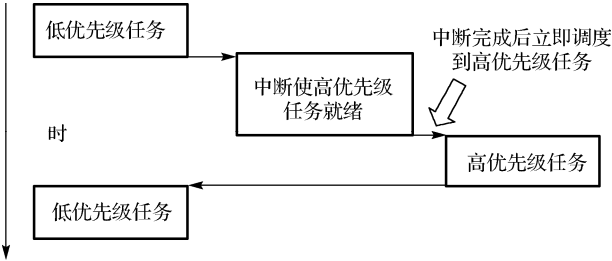


图 7-7 可剥夺型内核的调度

每个任务按其重要性相应地赋予一定的优先级。在基于优先级的调度策略中，高优先级的任务会首先获得运行的权力，而较低优先级的任务即使就绪了，如果有高优先级任务同时就绪的话，系统会首先选择运行高优先级的任务。优先级的设置有两种：静态优先级与动态优先级。在应用程序执行过程中诸任务优先级不变，则称之为静态优先级。在静态优先级系统中，诸任务以及它们的时间约束在程序编译时是已知的。如果应用程序执行过程中，任务的优先级是可变的，则称之为动态优先级。

我们以图 7-8 来简单阐述一下基于优先级的任务调度。在任务就绪队列中有一系列优先级不同的就绪任务。在图中有一个优先级队列，从上至下优先级由高至低。每一个优先级后挂了几个就绪的任务。任务调度的顺序是由上而下，由左至右。每次调度时，OS 都是从第一级优先级开始查询，若有就绪的任务就执行；若没有就绪任务，则接着向下查询，直到找到一个优先级最高的就绪任务并将 CPU 交给此任务运行。在所有的任务中有一个空闲（idle）任务，这个任务一直处于就绪状态。在其他的任务都没有就绪时，系统运行此任务。也就是说 idle 任务的优先级是最低的。

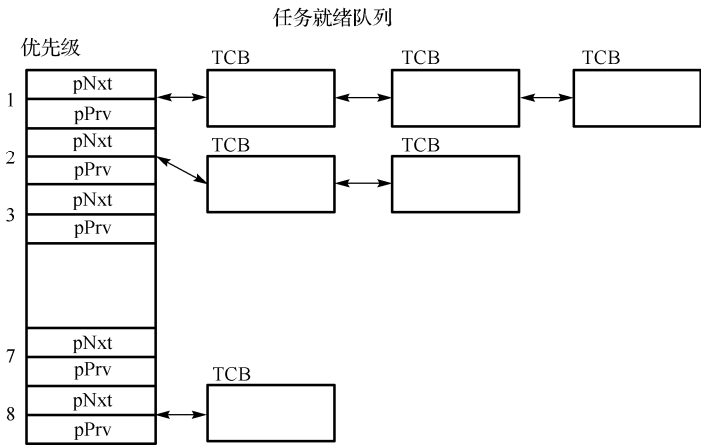


图 7-8 基于优先级的任务调度

5. 优先级反转

对于采用基于优先级调度的抢占式内核操作系统，都有可能引起所谓优先级反转的问题。如图 7-9 所示，A 任务的优先级比较低，其在运行过程中使用了临界资源 S，并在使用前为资源 S 加了锁，以保证在使用 S 的过程中不会被其他任务抢占。高优先级任务 C 也需要使用资源 S，由于 S 已被 A 加锁，因此 C 只能处于等待状态直到 A 释放该资源。在 T1 时刻，由于中断的发生，激活了任务 B，由于任务 B 的优先级高于 A，按照调度规则，内核将 A 挂起（变成就绪态）而运行 B。在 T2 时刻，另一个中断激活了一个更高优先级的任务 D，内核又将挂起 B 转而运行 D。总之，由于 A 被不断抢占，因此

A 也就无法尽快完成对临界资源 S 的使用, 从而使得高优先级任务 C 得到运行。造成的结果是优先级低于 C 的任务 B 和 D 却可以提前运行, 我们将这种现象称为优先级反转。

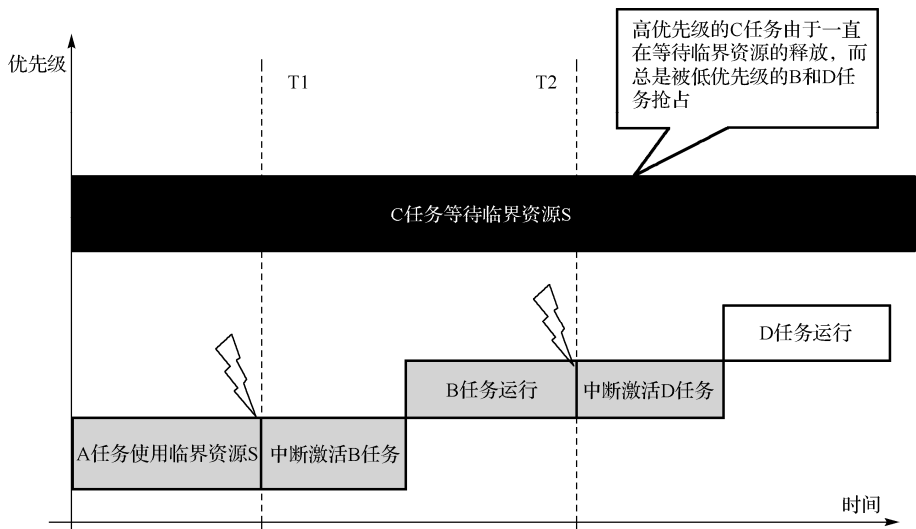


图 7-9 优先级反转

优先级反转是一个非常严重的问题, 它将造成基于优先级调度的失效。为了解决这个问题, 通常的做法之一称为优先级继承算法。如图 7-10 所示, 低优先级任务 A 正在运行并使用临界资源 S, 高优先级任务 C 抢占了 A 任务, 并试图也使用资源 S, 由于 S 已被 A 加锁, 操作系统内核将 C 转变为等待态。与没有实现优先级继承的操作系统不同的是, 内核发现目前占用临界资源的任务 A 优先级低于 C, 因此内核在挂起任务 C 的同时, 将 A 的优先级动态提升到与 C 的优先级一样 (这就是为什么称该算法为优先级继承的原因)。提升优先级后的 A 将继续运行, T1 时刻的中断激活了任务 B, 但由于此时 B 的优先级低于 A (因为 A 的优先级现在等于 C), 因此在中断完成后的 T2 时刻, A 将会继续被执行直到 T3 时刻, A 释放临界资源 S, 由于释放的过程是通过调用系统调用完成的, 因此内核释放资源 S 时, 将 A 的优先级恢复到原来的级别, 并在 T4 时刻将任务 C 调度运行 (因为 C 现在可以获得资源 S 了)。

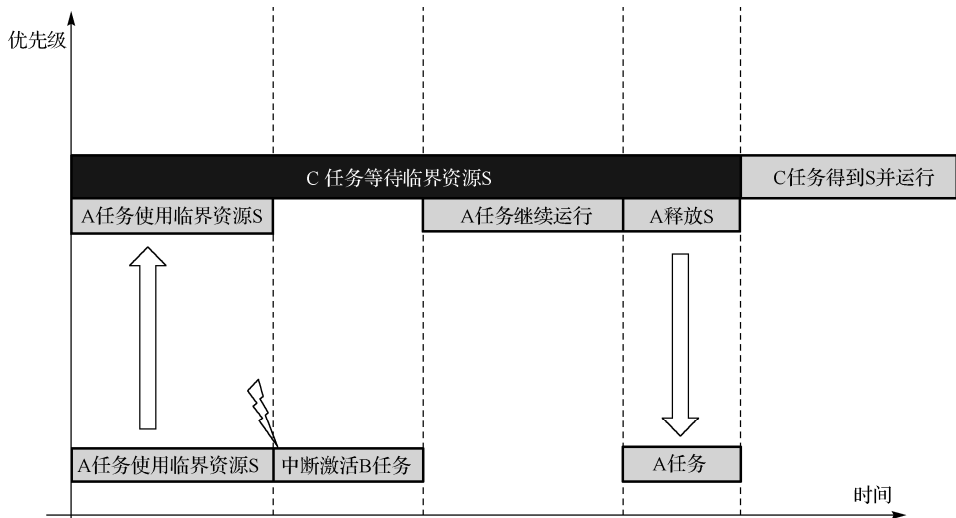


图 7-10 优先级继承算法

7.2.4 任务间通信

正如任何一个人类社会的组织，其所有成员间必须通过沟通才能够产生协作，在一个多任务系统中，任务之间往往需要通过传递信息以实现任务间的协同工作，包括对资源的并发访问、任务间的同步机制以及 Android 系统中采用的客户端-服务器模式等。我们把这种任务间的信息传递机制称为任务间通信。然而，在一个多任务并发的系统中，尤其是在采用中断实现抢占式多任务和进程空间保护的系统中，实现任务间的通信并不像想象的那么简单。操作系统必须为用户提供安全、高效、方便的任务间通信机制，本节将介绍几种常见的通信机制，在 7.3 节中还将介绍 Android 系统特有的一种任务通信机制。

1. 互斥与临界区

在介绍任务间通信之前，首先需要介绍临界资源（Critical Resource）的概念。所谓临界资源是指在任一时刻只能被独享而不能被分享的资源。一个关于临界资源的形象（虽然有点不雅观）的例子是高铁（或是飞机）上的洗手间，每个洗手间每次只能被一个乘客使用，首先进入洗手间的乘客需要在里面将门锁上，这时洗手间外的指示灯会告诉所有其他乘客目前该洗手间被占用了，需要使用的乘客只能在门外排队等待，等里面的乘客出来后，排在第一的乘客可以进入，然后他也会继续把门上锁……以此类推。在这个例子中，洗手间属于临界资源，该资源在任何一个时刻只能被一个用户使用，在该资源的使用过程中不能被打断，为了防止被意外打断，使用的乘客需要上锁。

在软件中，使用临界资源的代码称为临界段，有时也称为临界区，指处理时不可分割的代码（或者称为“重入”，我们将在 7.2.5 节介绍重入）。一旦这部分代码开始执行，则不允许其他代码使用该临界资源。临界资源可以是输入/输出设备，例如打印机、键盘、显示器，也可以是一个变量、一个结构或一个数组等。为了防止数据被破坏，每个任务在使用临界资源时必须独占，这称为互斥（mutual exclusion）。为确保临界区代码的安全执行，在进入临界区之前可以关中断（这是上锁的一种方法），临界区代码执行完以后要立即开中断。

通过关中断的方式对临界资源进行互斥保护是需要付出昂贵的代价的，尤其是如果临界区的代码比较多，因此关中断的时间会比较长的情况下更是如此。因为这意味着在关中断期间，系统将对所有的中断请求不再响应，这对于强实时的应用而言可能是灾难性的（试想汽车发生碰撞时，你的车载程序正运行在关中断的状态……）。基于上面的考虑，在工程实践中除了通过关中断来实现互斥外，通常的做法是使用信号量对临界区上锁或者通过暂时关闭任务调度器（前提是程序员需要保证在中断处理程序中不会访问该临界资源）。

2. 信号量

信号量实际上是一种约定机制，在多任务内核中普遍使用信号量用于：

- 控制共享资源的使用权（满足互斥条件）。
- 标志某事件的发生。
- 使两个任务的行为同步。

信号像是一把钥匙，任务要运行下去，要先拿到这把钥匙。如果信号已被别的其他任务占用，想再获得该钥匙的任务只得被挂起（也就是被操作系统从运行态切换为等待态），直到信号被当前使用者释放（想想我们的高铁洗手间模型）。换句话说，申请信号的任务是在说：“把钥匙给我，如果谁正在用着，我只好等！”。传统上，信号是只有两个值的变量：0 或者 1。1 表示当前信号量可用，0 则表示该信号量已被占用了。还有一种计数式信号量，它的值可以是 0~255，或 0~65 535，或 0~4 294 967 295，具体

取决于信号量规约机制使用的是 8 位、16 位还是 32 位。可以用多个隔间的公共洗手间比喻计数信号量，所有非零的取值表示可用的隔间数，每进入一个人，则信号量的值减 1，直到该值减为 0，则表示不能再有新用户进入了，新申请的用户必须等待（这时操作系统会把该任务从运行态调度为等待态）。在某些 CPU 架构上为了实现对信号量值的安全控制（加 1 或减 1），在写该变量前，操作系统需要关闭中断，写完后再次开中断。因此在这样的系统中，使用信号量也会关中断，只不过与用户自己开关中断相比，信号量关中断的时间非常短。而在某些 CPU 架构上，处理器提供了在一条指令中实现数据交换的指令，操作可以用该指令实现信号量的操作，这样就不需要关中断了（因为中断不可能打断一条指令的执行）。

一般来说，对信号量只能实施 3 种操作：初始化（INITIALIZE），也可称作建立（CREATE）；等信号（WAIT），也可称作挂起（PEND）；给信号（SIGNAL）或发信号（POST）。信号量初始化时要给信号量赋初值，等待信号量的任务表（Waiting list）应清为空。想要得到信号量的任务执行等待（WAIT）操作。如果该信号量有效（即信号量值大于 0），则信号量值减 1，任务得以继续运行。如果信号量的值为 0，等待信号量的任务就被列入等待信号量任务表。多数内核允许用户定义等待超时，如果等待时间超过了某一设定值时，该信号量还是无效，则等待信号量的任务进入就绪态准备运行，并返回出错代码（指出发生了等待超时错误）。

任务以发信号操作（SIGNAL）释放信号量。如果没有任务在等待信号量，信号量的值仅仅是简单地加 1。如果有任务在等待该信号量，那么就会有一个任务（通常是等待队列中最前面的那个任务）从等待态转为就绪态，信号量的值也就不加 1。收到信号量的任务可能是以下两者之一：

- 等待信号量任务中优先级最高的；
- 最早开始等待信号量的那个任务，即按先进先出的原则（FIFO, First In First Out）。

内核有选择项，允许用户在信号量初始化时选定上述两种方法中的一种。如果进入就绪态的任务比当前运行的任务优先级高（假设是当前任务释放的信号量激活了比自己优先级高的任务），则内核做任务切换（假设使用的是可剥夺型内核），高优先级的任务开始运行，而当前任务被挂起（就绪态）。

3. 消息邮箱

通过内核服务一个任务可以给另外一个任务发送消息。典型的消息邮箱也称作交换消息，是用一个指针型变量，通过内核服务、一个任务或一个中断服务程序可以把一则消息（即一个指针）放到邮箱（系统通过这个邮箱的名字来进行标识，这个名字通常是创建这个邮箱时指定的）中。同样，一个或多个任务可以通过内核服务接收这则消息。发送消息的任务和接收消息的任务约定，该指针指向的内容就是那则消息。

每个邮箱有相应的正在等待发往本邮箱消息的任务队列，试图从该邮箱收取消息的任务会因为邮箱是空的而被内核挂起（从运行态切换为等待态），并加入到该等待任务队列中，直到邮箱收到消息时内核将唤醒队列中的第一个任务。一般来说，内核允许用户定义等待超时，若等待消息的时间超过了仍然没有收到该消息，该任务将被内核切换到就绪态，并返回出错信息，报告等待超时错误。消息放入邮箱后，或者是把消息传给等待消息的任务表中优先级最高的那个任务（基于优先级），或者是将消息传给最先开始等待消息的任务（基于先进先出原则）。图 7-11 示意把消息放入邮箱。用一个 I 字表示邮箱，旁边的小砂漏表示超时计时器，计时器旁边的数字表示定时器设定值，即任务最长可以等多少个时钟节拍（Clock Ticks）。

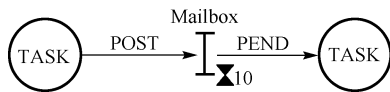


图 7-11 消息邮箱

内核一般提供以下邮箱服务：

- 邮箱内消息的内容初始化，邮箱里最初可以有、也可以没有消息。

- 将消息放入邮箱（POST）。
- 等待有消息进入邮箱（PEND）。

消息邮箱也可以当作只取两个值的信号量来用。邮箱里有消息，表示资源可以使用，而空邮箱表示资源已被其他任务占用。

4. 消息队列

消息队列用于给任务发消息。消息队列实际上是邮箱阵列。通过内核提供的服务，任务或中断服务子程序可以将一条消息（该消息的指针）放入消息队列。同样，一个或多个任务可以通过内核服务从消息队列中得到消息。发送和接收消息的任务约定，传递消息实际上是传递指针指向的内容。通常，先进入消息队列的消息先传给任务，也就是说，任务先得到的是最先进入消息队列的消息，即先进先出原则（FIFO）。

像使用邮箱那样，当一个以上的任务要从消息队列接收消息时，每个消息队列有一张等待消息任务的等待列表（Waiting List）。如果消息队列中没有消息，即消息队列是空，等待消息的任务就被挂起并放入等待消息任务列表中，直到有消息到来。通常，内核允许等待消息的任务定义等待超时的时间。如果限定时间内任务没有收到消息，该任务就进入就绪态并开始运行，同时返回出错代码，指出出现等待超时错误。一旦一则消息放入消息队列，该消息将传给等待消息的任务中优先级最高的那个任务，或是最先进入等待消息任务列表的任务。图 7-12

示意中断服务子程序如何将消息放入消息队列。图中两个大写的 I 表示消息队列，“10”表示消息队列最多可以放 10 条消息，沙漏旁边的 0 表示任务没有定义超时，将永远等下去，直至消息到来。

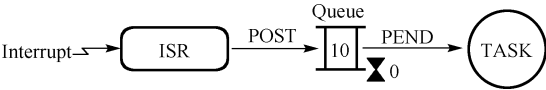


图 7-12 消息队列

- 典型地，内核提供的消息队列服务如下：
- 消息队列初始化。队列初始化时总是清为空。
 - 放一则消息到队列中（POST）。
 - 等待一则消息的到来（PEND）。

如果队列中有消息则任务可以得到消息，但如果此时队列为空，内核并不将该任务挂起（Accept）。如果有消息，则消息从队列中取走。没有消息则用特别的返回代码通知调用者，队列中没有消息。

7.2.5 中断管理

我们一般将中断分为硬件中断、软件中断和异常 3 类，尽管这些中断产生的原因各有不同，硬件和操作系统处理它们的过程却是基本相同的，本书在 4.2.5 节比较详细地介绍了嵌入式微处理器对于中断的处理过程。总的来说，中断是异步的突发事件，为了高效而安全地处理这些异步事件，处理器和操作系统必须紧密配合。

“中断是操作系统的入口”，这是出自塔利鲍姆《操作系统设计与实现》中的一句话。在理解了操作系统的基本原理后，我们会发现大师的总结实在是太精辟了。几乎所有的操作系统都要全权接管中断，任务的调度需要依靠中断，系统调用的实现也需要依靠中断，可以不夸张地认为，整个操作系统都是建立在中断处理的基础之上的。本节将介绍中断服务程序的堆栈组织、与中断相关的函数重入问题以及操作系统中的中断处理程序。

1. 栈帧管理

堆栈对于所有的计算机软件而言都是非常重要的，正如 7.2.3 节所介绍的，程序执行流利用堆栈

保存所有的函数调用顺序,使得所有的被调用函数(通常称为 Callee)在执行完毕后能够通过堆栈中保存的返回地址将程序返回到调用函数(通常称为 Caller)处。除了保存返回地址外,编译器通常还利用堆栈完成其他另外 3 个功能:传递函数调用参数;实现 Callee 中使用的临时变量(编译器首先尽可能使用 CPU 内的寄存器实现这些临时变量,对于数组或其他无法用寄存器实现的临时变量则通过堆栈空间实现);对于 Callee 临时变量所使用的寄存器,如果它们也被 Caller 使用,则编译器还需要利用堆栈保存 Caller 使用寄存器的值,并在 Callee 退出时将这些值退栈到寄存器中。一般我们把由于一次函数调用而构建的栈空间称为函数调用栈帧,如图 7-13 所示。

```
U32 func1(U32 arg1, void *ptr, U16 arg3);
main()
{
    U32 y;
    :
    I = func1(a, p, c);
    :
}

U32 func1(U32 arg1, void *ptr, U16 arg3)
{
    U32 x;
    U32 LocalArray[10];
    :
    Return x;
}
```

在前面的代码中, U32 是我们定义的 32 位无符号整数。Caller 指 main 函数, Callee 指 func1 被调用函数。main 函数在执行到 `I = func1(a, p, c)` 时,编译器的压栈过程为(从高地址到低地址压栈):

(1) 先压参数。对于 Motorola 的 68K 系列处理器来说,编译器对标准的 C 语言中参数的压栈顺序是从右向左,先压参数 `c`,然后是 `p`,最后 `a`。ARM 的编译器对参数传递的规则根据 4.2.6 节中介绍的 ATPCS 规则约定,如果参数少于 4 个,编译器利用 `r0~r3` 来传递参数,在这个例子中有 3 个参数,所以是用 `r0~r2` 来传递参数的,而不是把参数压入堆栈中,如果参数超过 4 个,则需要用堆栈来传递参数。

(2) 再压程序的返回地址。也就是 Caller 中调用子程序之后的下一条指令的地址。这一步对于 68K 系列处理器和 ARM 处理器是一样的。ARM 在子程序调用时会把返回地址存入 `r14` 即 LR 连接寄存器中,这是硬件自动完成的,接下来要通过软件把 `r14` 的值保存到堆栈中,这是由编译器完成的。

(3) 将 Callee 中可能用到的寄存器的值保存起来。本例中 main 函数(Caller)使用了一个局部变量 U32 `y`,通常 `y` 会被一个寄存器所表示,由于 func1 函数中也用到了一个局部变量 U32 `x`,如果编译器试图将 `x` 采用与 `y` 相同的寄存器来保存的话,则需要首先将该寄存器中保存的 `y` 的值先保存在堆栈中,在返回 Caller 之前再将保存的 `y` 值退栈到寄存器中。在图 7-13 中,我们用 Saved Regs 表示这些保存的寄存器值。

(4) 最后,编译器将把 Callee 中的一些无法用寄存器实现的局部变量保存在堆栈中。比如在上面

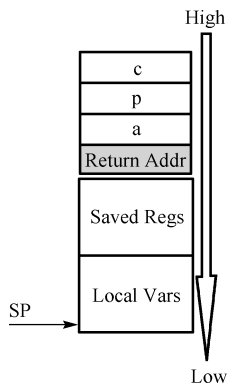


图 7-13 函数调用栈帧

的代码中，Callee 声明了一个整形数组 LocalArray[10]，显然编译器没有办法利用寄存器实现这个有 10 个元素的数组，只能将其保存在堆栈中，图 7-13 中表示为 Local Vars。函数的返回值 x 存放在 r0 中，由 r0 来传递给被调函数。

不同的编译器对函数调用堆栈的处理不完全相同，但是大同小异。在不同的编译器中往往规定不同寄存器的不同用途，有些寄存器可以指定用作存储返回值、返回地址、参数、临时变量等。

与一般的函数调用栈帧不同，中断栈帧没有参数传递，因为中断服务程序没有入口参数。另一点重要不同是除了需要保存返回地址外，我们还需要将中断前的程序状态寄存器的内容保存到堆栈中，这样在从中断返回的时候不仅要恢复返回地址，也需要恢复程序状态寄存器的值。对于很多 CISC 处理器而言，返回地址与程序状态子 PSR 是硬件在响应中断的时候自动保存到堆栈中的，而对于采用 Load/Store 架构的 RISC 处理器，比如 ARM，在响应中断时硬件只是自动将返回地址和程序状态字保存到 r14 寄存器和 SPSR 寄存器中，程序员需要通过汇编代码显式地将 r14 与 SPSR 中的内容压栈。中断服务程序还需要将需要用的寄存器压栈，如果中断服务程序需要使用的局部变量无法用寄存器表示，则还需要在堆栈中开设专门的区域用于保存这些变量。中断栈帧的一般组织如图 7-14 所示。

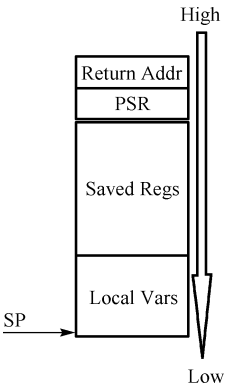


图 7-14 中断栈帧

函数调用栈帧与中断栈帧的这种区别对于操作系统的实现是非常重要的，我们在后面会介绍到操作系统的调度只会发生在两个地方：一是系统调用结束之前，因为系统调用的执行有可能改变任务的状态（比如一个访问信号量的系统调用可能将调用该系统调用的任务由运行态变成等待态，这时就需要内核调度一个新的任务进入运行态）；二是中断服务程序返回之前，因为中断的发生也可能改变任务的状态（当然，这要求中断服务程序显式地调用相关系统调用来改变任务状态）。如果操作系统的系统调用通过软陷来实现，也就是通过软中断的服务程序来实现的话，那么系统调用结束前的栈帧组织和中断服务程序结束前的栈帧组织是相同的（都属于中断栈帧），内核在处理调度过程中的栈操作就是统一的。另外一种情况是系统调用采用函数调用的方式实现，这种情况对于没有 MMU 和用户模式与特权模式之分的系统而言非常常见（比如 uCOS/II），内核在处理系统调用（其实在这些系统中系统调用已退化为由内核提供的一组函数）和中断返回处的栈帧是不一样的，因为前者是调用栈而后者是中断栈。

2. 函数的重入

函数的重入对于多任务系统而言是经常出现的问题，所谓函数的重入是指一个函数被多个执行流进入。这个定义不是非常好理解，我们将用图 7-15 进行说明。函数的重入只会在 3 种情况下发生，分别是中断引起的重入、多任务引起的重入和递归引起的重入。

首先是中断引起的函数重入。如图 7-15(a)所示，一个任务正在执行函数 Func A，在该函数执行完毕前，由于中断 IRQ 的发生，处理器执行中断服务程序 ISR，在该中断服务程序中又一次调用了函数 Func A，这样就发生了执行流第二次进入该函数，我们称其为由于中断而引起的重入。图 7-15(b)所示是由于另一个任务而引起的重入，Task1 正在执行函数 Func A，由于中断 IRQ 的发生在中断服务程序 ISR 中激活了另一个高优先级任务 Task2，而 Task2 调用了函数 Func A，这也引起了函数 Func A 的重入。第三种情况是由于递归引起的函数重入，这种情况相对比较简单，就是函数 Func A 在执行完毕前又调用了自己（这也是递归的定义）。递归引起的重入是程序员通过程序显式实现的，程序员知道什么时候会发生重入，因此相对来说这种情况的重入通常是安全的。

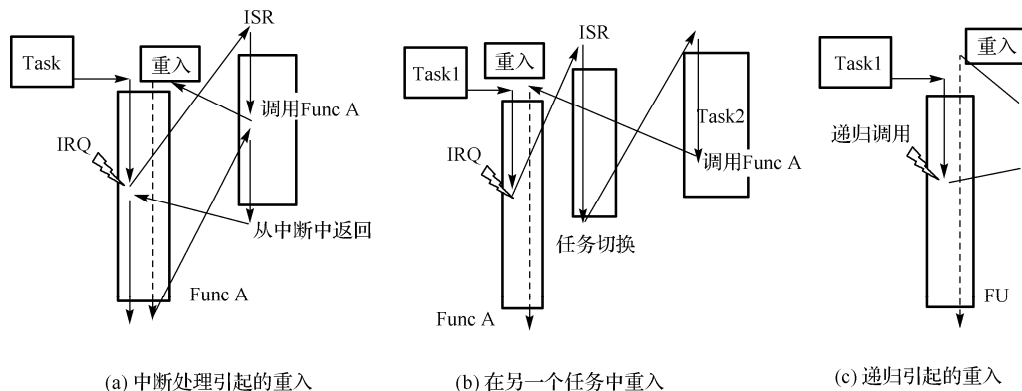


图 7-15 函数的重入

然而，并不是所有的函数都可以安全重入。所谓安全重入是指，函数的重入并不会引起错误。引起函数重入错误的根本原因是函数中使用了临界资源，而临界资源是不能够被分享的。这些临界资源最主要体现为全局数据或者调用了其他不可安全重入的函数。由于第一种重入是由于中断服务程序调用该函数引起的，所以如果我们在调用该函数前关中断，而在出该函数后再恢复中断，就可以保证不会发生重入了。对于第二种情况，在调用该函数前，关闭内核调度，使得在完成该函数前，操作系统不会发生任务调度，也可以保证该函数不发生重入（除非中断处理程序中调用该函数）。其实更简洁的方法是使用信号量来解决函数的重入问题，因为引起不安全重入的根本原因是函数中使用了诸如全局变量这样的临界资源，因此只需要在使用这些临界资源前使用信号量加锁操作，在使用完成后再进行解锁操作就可以很好地实现临界资源的互斥操作了（当然这种方法还需要考虑中断处理程序中是否会调用该函数的问题）。

3. 操作系统中的中断处理程序

几乎所有的操作系统都全权接管对中断的管理，包括：所有的外部硬件中断，以处理异步的外部 I/O 事件；所有的软件中断，以实现系统调用；所有的异常，以实现虚拟内存管理和其他的内部错误处理。操作系统通过接管系统的中断向量表来实现对所有中断的接管，如下面的代码来自国家 ASIC 工程中心研发的 ASIX OS，该段代码定义了中断向量表，其中 Irq_Do 函数是系统处理硬件中断的入口。

```

; /*****
;   file name :   boot.s
;   description: boot the arm processor
;   history:      ****
; *****/
include hardware_gfd.h

extern main
AREA BOOT, CODE, READONLY
ENTRY                                ;第一条指令从这里开始
;vector table
    bal    RST_DO
    bal    EXTENT_INSTRU
    bal    SWI_DO
    bal    ABORT_PREFETCH_DO
    bal    ABORT_DATA_DO
    mov    R1, R1                                ;保留中断

```

```

    bal    Irq_Do
    mov    r0, r0
    bal    Fiq_Do                ;快速中断 FIQ 的处理

```

硬件在响应中断后会执行 `Irq_Do` 函数，该函数由汇编语言编写，主要的工作包括返回地址与程序状态字入栈（对于 ARM 处理器而言这步工作必须由程序员通过 `Store` 指令显式完成）。另外，该程序还需要处理中断嵌套的问题和寄存器值保存的相关问题。在处理完这些工作后，`Irq_Do` 函数将调用 C 语言编写的中断处理函数 `int_vector_handler`。该函数首先读取中断控制器硬件的状态寄存器，并通过一个循环获得具体的中断源标号。（代码中以变量 `i` 表示。）系统中维护了一张记录所有中断处理函数的函数指针数组 `IntHandler[]`，在这个数组中以中断源标号为索引，记录了相对应的中断处理函数的指针，因此在 `int_vector_handler` 函数中只需要通过语句“`if (IntHandler[i]) (*IntHandler[i]) ();`”就可以首先判断该处理函数不为空的情况下，调用相应的中断处理函数。

```

void (*IntHandler[32])(void)={
    /*interrupt number and description, handler */
    /* 00 INT_NULL,          */          ENT_INT_EMPTY      ,
    /* 01 INT_EXT0,  (PE0) */          ENT_INT_RING1       ,
    /* 02 INT_EXT1,  (PE1) */          NULL                  ,
    /* 03 INT_EXT2,  (PE2) */          NULL                  ,
    /* 04 INT_EXT3,  (PE3) */          ENT_INT_RING2       ,
    /* 05 INT_EXT4,  (PE4) */          NULL                  ,
    /* 06 INT_EXT5,  (PE5) */          NULL                  ,
    /* 07 INT_EXT6,  (PE6) */          ENT_INT_RING3       ,
    /* 08 INT_EXT7,  (PE7) */          NULL                  ,
    /* 09 INT_EXT8,  (PE8) */          NULL                  ,
    /* 10 INT_EXT9,  (PE9 */          ENT_INT_BUTTON       ,
    /* 11 INT_EXT10, (PE10)*/          NULL                  ,
    /* 12 INT_EXT11, (PE11)*/          NULL                  ,
    /* 13 INT_EXT12, (PH0) */          NULL                  ,
    /* 14 INT_EXT13, (PH1) */          ENT_INT_SSRT         ,
    /* 15 INT_EXT14, (PH2) */          NULL                  ,
    /* 16 INT_NONE,    */          NULL                     ,
    /* 17 INT_EXT15,  (PH3) */          NULL                  ,
    /* 18 INT_EXT16,  (PH4) */          NULL                  ,
    /* 19 INT_EXT17,  (PH5) */          NULL                  ,
    /* 20 INT_LCDC,    */          NULL                     ,
    /* 21 INT_AC97,    */          NULL                     ,
    /* 22 INT_PWM,     */          NULL                     ,
    /* 23 INT_UART1,   */          NULL                     ,
    /* 24 INT_UART0,   */          NULL                     ,
    /* 25 INT_MMC,     */          NULL                     ,
    /* 26 INT_SPI,     */          NULL                     ,
    /* 27 INT_USB,     */          ENT_INT_USB              ,
    /* 28 INT_GPT,     */          ENT_INT_GPT              ,
    /* 29 INT_EMI,     */          ENT_INT_EMI              ,
    /* 30 INT_DMA,     */          ENT_INT_DMA              ,
    /* 31 INT_RTC,     */          ENT_INT_RTC              ,
};

void int_vector_handler(void)

```

```

{
int i;

//读中断控制器的相关状态寄存器
unsigned long IFSTAT=*(RP) (INTC_IFSTAT);
if (IFSTAT<1) IFSTAT=*(RP) (INTC_ISTAT)& (~*(RP) (INTC_IMSK))& *(RP) (INTC_IEN);
//查找中断控制器中状态寄存器的置位，以确定中断源

    if (IFSTAT>1)
    {
        i =-1;
        while (IFSTAT)
        {
            IFSTAT>>=1;

            i++;
        }
    }else i =0;

    //如果对应的中断源的处理函数不为空，则调用该函数处理中断
    if (IntHandler[i]) (*IntHandler[i]) ();
else
{
    //否则报错
    ent_int();
    printf("No interrupt entry for INT NO. %d\n", i);
    ret_int();
}
}

```

如果所发生的中断没有对应的处理函数，则系统通过 `printf` 函数^①报错。需要注意的是在调用 `printf` 之前，首先调用了 `ent_int` 函数，而在执行完 `printf` 之后则调用了 `ret_int` 函数。ASIX OS 通过这两个函数对中断处理程序进行管理。函数 `ent_int` 的主要功能是通知内核现在处理器进入了中断处理程序，并记录中断嵌套的层次（如果系统允许中断嵌套的话）；而 `ret_int` 函数的主要功能则是通知内核即将离开中断处理程序，然后判断是否依然在嵌套的中断处理程序中。如果依然在嵌套中，则恢复相应的中断栈帧返回到上一级中断处理程序；否则我们即将离开最后一层中断处理程序，在这之前 `ret_int` 将调用 `scheduler` 函数，对系统内的任务进行一次新的调度。从上面的分析可以看出 `ent_int` 和 `ret_int` 函数的重要作用，因此所有的中断服务程序在进入时应该显式调用 `ent_int`，而在离开前应该相应地调用 `ret_int`。程序中在调用 `(*IntHandler[i])()` 前后没有调用这两个函数，是希望编写中断服务程序的程序员在其函数中自行调用。当然上面的代码也可以改为：

```

:
if (IntHandler[i]) {
    ent_int();
    (*IntHandler[i]) ();
    ret_int();
} else{

```

① ANSI C 标准并没有要求 `printf` 函数的实现是可安全重入的，由于这里的 `printf` 函数调用发生在中断处理程序之中，因此程序员需要确保这样的调用是安全的。

⋮
}

ASIX OS 没有采用这种强制要求中断服务程序调用 `ent_int` 和 `ret_int` 的原因是把是否调用的主动权留给中断服务程序的编写人员，在某些极端情况下，为了更快地响应和处理中断，程序员可以通过不调用这两个函数而绕开内核监管，自行管理中断（代价是这个自行管理的中断服务程序不能调用任何操作系统系统调用，也不能影响任何任务的运行状态）。

不同的操作系统管理中断的方式各不相同，但基本原理应该是基本相同的，通过研读相关系统的源码可以深入地理解操作系统是如何管理中断的。

4. 系统调用与函数调用

系统调用从用户的角度看与函数调用没有什么区别，但实际上系统调用一般都是在内部通过软件中断的方法使 CPU 进入特权状态，从而实现对全部资源的访问。正如第 4 章中所介绍的，CPU 内核一旦切换到用户态，回到特权模式的唯一方法就是通过中断。采用软件中断实现系统调用的另外一个优势正如前面所介绍的，系统调用构建的栈帧就是中断栈帧，这使得调度器在管理任务上下文切换时可以采用统一的方式进行堆栈管理。

当然，采用软件中断（有时称为软陷）的方式实现系统调用也是有代价的。通常情况下，CPU 和软件系统在响应中断时需要花费更多的时间来进行寄存器值的压栈操作（所谓保护现场）。同样，在中断返回时必须将这些保存的寄存器值退栈到相应的寄存器中。这些压栈和退栈操作都是访存操作，在第 5 章我们介绍过现代高性能嵌入式微处理器访问外存的延时可能是访问内部寄存器的 100 倍以上，这将大大增加中断处理的开销。也正是因为这个原因，在很多不需要严格保护的深嵌入式系统中，操作系统会提供基于函数调用的 API，这将使得系统调用的开销大大减少。

7.3* Android 操作系统简介

随着通信技术的高速发展，移动设备已经成为人们工作和生活中必不可少的重要工具。科技的发展使得人们对嵌入式操作系统有了更强烈的需求，在此趋势下出现了许多优秀的操作系统。例如，谷歌公司的 Android，苹果公司的 iOS，微软公司的 Windows Phone，RIM 公司的 Blackberry，等等。其中 Android 系统脱颖而出，占领了智能手机市场的绝大部分份额。2011 年第一季度，Android 在全球的市场份额首次超过塞班系统，跃居全球第一。在 2014 年 Google I/O 开发者大会上 Google 宣布过去 30 天里有 10 亿台

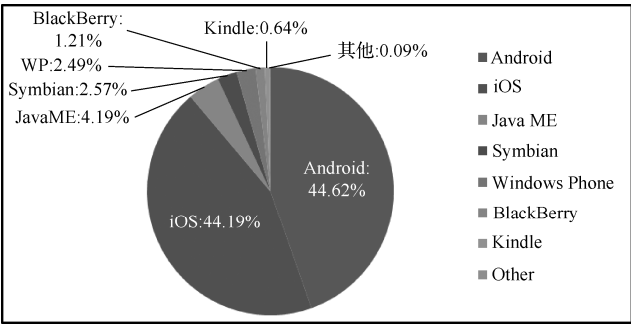


图 7-16 2014 年 7 月各大移动平台的市场占有率

Android 设备被激活。根据著名互联网流量监测机构 Net Application 发布的数据，Android 市场占有率一直占据市场领先地位。如图 7-16 所示为各大移动平台的分布情况。

7.3.1 Android 操作系统的层次

Android 是一种基于 Linux 的自由及开放源代码的操作系统，由 Google 公司和开放手机联盟领导及开发，其系统架构如图 7-17 所示。

从图 7-17 中可以看出，Android 系统架构由上到下主要包含 4 层，分别介绍如下。

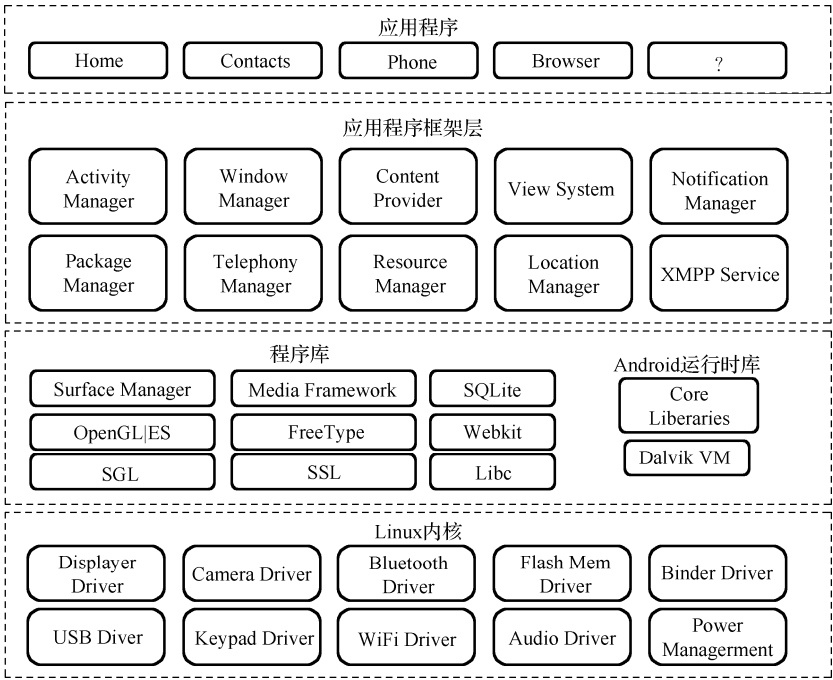


图 7-17 Android 系统结构图

1. 应用层

应用层是直接为用户进行交互的一层，包括各种具有用户界面（UI，User Interface）、提供一定功能的应用程序（如启动器、拨号程序、短信应用等）。该层应用程序使用 Java 语言编写，应用层内所有程序地位平等，开发者可以使用自己编写的程序对系统应用进行替换，这个替换由应用程序框架层保证。

2. 应用程序框架层

这一层是 Google 在开发核心应用程序时使用的 API 框架，开发者可以利用这些框架进行第三方应用程序的开发，但是在开发时必须遵守框架规定的开发原则和安全性限制。表 7-1 对框架层中提供的主要组件和功能进行了说明。

表 7-1 应用程序框架层提供的组件及其功能描述

组件	功能描述
活动管理器 (Activity Manager)	管理应用程序的生命周期并提供常用的导航回退功能，为所有程序的窗口提供交互的接口
视图系统 (View System)	创建应用程序的基本组件，包括列表（list）、网格（grid）、文本框（text boxes）、按钮（button），还有嵌入式的 Web 浏览器
窗口管理器 (Window Manager)	对所有开启的应用程序窗口进行管理
通知管理器 (Notification Manager)	使应用程序可以在状态栏中显示自定义的提示信息
资源管理器 (Resource Manager)	提供各种非代码资源供应用程序使用，如本地化字符串、图片、音频等
挂载服务 (Mount Service)	管理和控制外部存储设备，包括 SDcard 的拔插、挂载、卸载、格式化等

续表

组件	功能描述
内容提供者 (Content Provider)	提供一个程序访问另一个应用程序数据的功能，或者实现应用程序之间的数据共享

3. 系统运行层

该层主要包含两大部分：程序库和 Android 运行时库。

(1) 程序库

程序库实际上是一些 C/C++库，这些库文件提供了许多功能，能够直接被 Android 系统的不同应用程序调用，并通过应用程序框架层给开发者提供服务。表 7-2 给出了部分程序库的功能描述。

表 7-2 程序库包含的核心库文件及其功能描述

名称	功能描述
Surface Manager	主要对多个应用程序同时执行进行管理，以及程序的显示与存取，并且为多个应用程序提供 2D 和 3D 的无缝融合
Media Framework	基于 OpenCORE，支持多种常用音视频格式的回放和录制
SQLite	一个功能强大的轻量级关系型数据库引擎
Webkit	网页浏览器引擎，用于支持 Android 浏览器和嵌入式的 Web 视图
OpenGL ES	同时支持 1.x 和 2.0 标准，该库可以使用硬件 3D 加速或者使用高度优化的 3D 软加速
FreeType	提供位图 (bitmap) 和矢量图 (vector) 两种文字体的显示
Libc	从 BSD 继承过来的标准 C 系统函数库，专门为基于嵌入式 Linux 的移动设备进行了定制

(2) Android 运行时库。

Android Runtime 又分为两大部分：Core Libraries 和 Dalvik VM。Core Libraries 是核心库的集合，提供了 Java 语言核心库的绝大多数功能，同时也提供了 Android 核心库。Dalvik VM 是 Android 平台的一个虚拟机，相当于 PC 中 Java 的虚拟机 JVM，但是 JVM 基于栈而 Dalvik 虚拟机是基于寄存器的，后者还能在同一设备上同时运行多个虚拟机，与前者相比更加节省系统资源。每一个应用层的 Android 应用都是在 Dalvik 虚拟机里独立运行在自己进程中的一个实例。为了保证在资源稀缺的移动设备上顺畅运行，Dalvik 虚拟机被优化成占用最小内存的 Dalvik Executable (.dex) 格式来执行文件。

4. Android 内核层

Android 内核是基于 Linux 定制的，除了修改部分 BUG 外，Google 还在其中加入了用于支持 Android 平台的设备驱动，主要包括提供进程间通信的 Android Binder，针对移动设备特别优化的 Power Manage 驱动，用于释放内存的低内存管理器 (Low Memory Killer)，匿名共享内存 (Ashmem) 和全新的内存分配器 (Android PMEM) 等。该层作为硬件和软件之间的抽象层，隐藏具体的硬件细节为上层提供统一的服务。

7.3.2 Android 虚拟机

Google 公司最初使用的 Android 虚拟机是由 Dan Bornstein 等人开发的 Dalvik 虚拟机。Dalvik 虚拟机的执行引擎由对字节码进行逐条解释执行的解释器和基于路径的即时编译器构成。Dalvik 虚拟机设计之初，面向的对象是硬件资源相对较弱的嵌入式设备，但随着微电子技术的发展，硬件平台性能的提升对软件平台提出了新的要求。Google 在 Android 4.4 中加入了试用版本的 ART 虚拟机运行时来应对当今高性能嵌入式设备的需求，并于 Android 5.0 完全取代 Dalvik 虚拟机。ART 虚拟机采用预编译器将字节码预编译为本地代码，应用程序运行时将直接执行本地代码，大大提高了 Java 程序运行速度。本节主要介绍 Dalvik 虚拟机和 ART 虚拟机的结构与特点。

Android 虚拟机运行由 Java 语言编写的应用程序，其本质上也是一个 Java 虚拟机。Android 虚拟机与标准的 Java 虚拟机具有许多相同特征，例如解释执行应用程序、垃圾回收机制、Java 本地方法调用等。但 Android 虚拟机并没有完全遵守 Java 虚拟机的规范，为支持硬件资源有限的移动设备，Android 虚拟机做了许多优化，具有一些有别于标准 Java 虚拟机的特征，主要表现在以下方面：

(1) 标准 Java 虚拟机基于栈架构，而 Android 虚拟机采用基于寄存器的架构。基于寄存器的指令需要指定源地址和目的地址，因此每条指令占用空间变大了，但总体上减少了数据读写时的指令分派和内存访问次数。

(2) 标准 Java 虚拟机运行的是 Java 字节码，Android 虚拟机运行的是 Java 字节码通过 dx 工具转化生成的 Dalvik 字节码。

(3) Android 虚拟机执行的是多个 Java 类文件重新编排，消除冗余信息，优化得到的 Dex 文件。减小应用程序文件尺寸的同时，提高了类查找的速度。

(4) 每个应用程序都有独立的 Android 虚拟机进程，避免一个应用程序进程崩溃影响其他应用程序。

(5) 使用只读的内存映射方式加载 Dex 文件，虚拟机间共享应用程序代码和数据，同样的内容仅被加载一次，减少性能开销。

1. Dalvik 虚拟机

Dalvik 虚拟机是 Google 公司为 Android 平台开发的虚拟机，是 Android 移动设备的核心组成部分之一。如图 7-18 所示，Dalvik 虚拟机由解释器、即时编译器、类加载器、运行时数据区和垃圾回收器等部分构成。

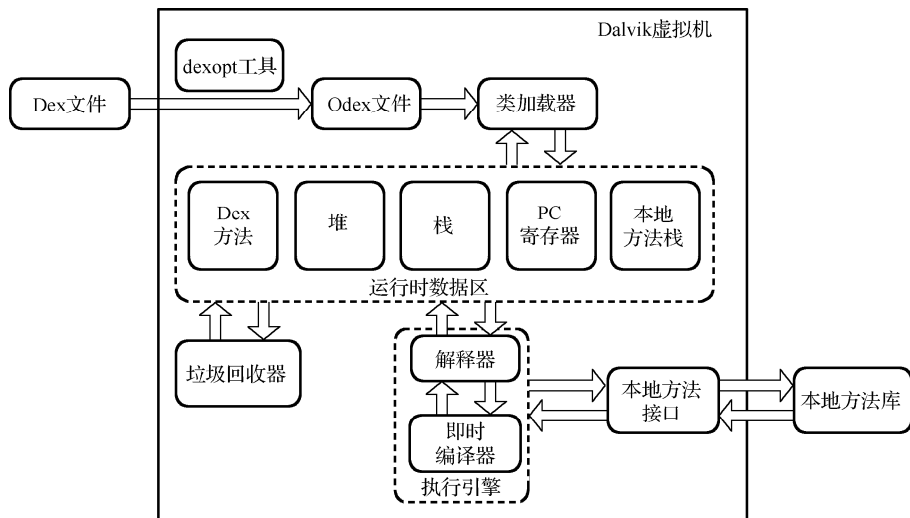


图 7-18 Dalvik 虚拟机架构图

Dalvik 虚拟机中包含的类加载器有用 C 语言编写的启动类加载器和 Java 语言编写的其他类加载器两种。启动类加载器用于 Dalvik 虚拟机初始化时加载一些 Dalvik 虚拟机运行时所需的 Java 类；其他类加载器用于程序运行时加载程序运行所需的 Java 类。运行时数据区用于存储 Android 系统运行过程中所需的数据，主要包括存储 Dex 文件中方法字节码和类描述信息的 Dex 方法区、存储类实例化对象的堆、存储基本类型变量的栈、存储 Dalvik 虚拟相关寄存器的寄存器区和为 Java 本地方法调用提供环境的本地方法栈。为了方便 Java 程序员开发，与标准 Java 虚拟机类似，Dalvik 虚拟机采用垃圾回收器来管理类对象占用的内存空间。

Dalvik 虚拟机中用于解析字节码的执行引擎，有解释器和即时编译器两个组成部分。解释器对字节码逐条解释执行，占用系统资源少，但效率较低；即时编译器通过对字节码进行筛选，对通过筛选的热点字节码段进行即时编译，再次执行相同的字节码段时直接运行本地代码，提高了程序运行速度。

Java 本地方法接口为 Java 语言与其他语言交互提供了基础，是 Dalvik 虚拟机的主要组成部分。Java 本地方法接口使 Dalvik 既可以运行由 Java 语言编写的应用程序，也可以执行应用程序包中由 C 语言编写的本地方法。

图 7-19 所示是 Dalvik 虚拟机中各模块间的关系。

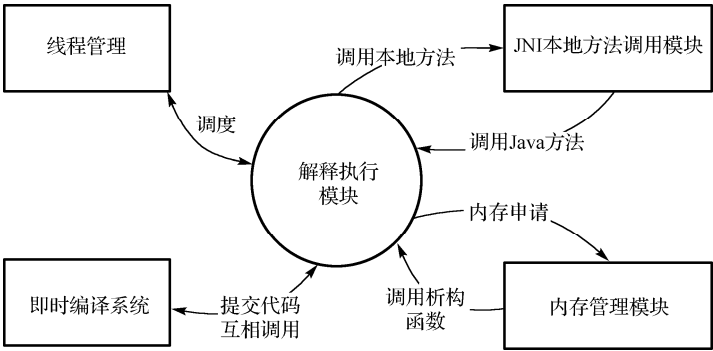


图 7-19 Dalvik 虚拟机各模块间的关系

2. Dalvik 字节码与 dex 文件

Android 应用程序使用 Java 语言编写，并在 Dalvik 虚拟机中运行。Dalvik 虚拟机有着不同于 Java 虚拟机的机制，它专门针对 Android 平台进行了特定的优化。通过 dx 的转换工具，可以将传统的 JVM 虚拟机中的字节码（.class 格式文件）转换为 Dalvik 虚拟机中运行的字节码（.dex 格式文件）。Java 字节码由一个或者多个.class 文件组成。在运行时，JVM 会动态地从各个.class 文件中将字节码加载到内存中。而 Dalvik 字节码只由一个.dex 文件构成，这个文件里包含了应用程序需要的所有类，图 7-20 展示了 Android 应用程序文件的生成过程。

如图 7-20 所示，由 Java 语言编写的 Android 应用程序源文件经 Java 编译器编译生成*.class 文件，并将其打包成*.jar 文件，再由 dx 工具将*.jar 文件生成*.dex 文件，即包含 Dalvik 字节码的文件，再由 odex 工具优化*.dex 文件，优化后生成*.odex 文件，最后将*.dex 文件或*.odex 文件和一些资源信息打包成*.apk 文件。从图中可以看出，dx 工具将多个*.class 文件整合，将数据段、常量池等归类放到一起，利用数据共享来减少内存开销。Dalvik 字节码是 Dalvik 虚拟机的指令集，Dalvik 字节码模拟真实处理器指令集架构，DVM 与 JVM 相比具有以下特点。

- （1）寄存器架构：不同于 JVM 基于栈处理器，DVM 基于寄存器处理器。在 Java 字节码中，本地变量存放在本地变量列表中，然后被压入堆栈，由操作码进行操作。或者，JVM 也可以不强制将本地变量存入本地变量列表中，而是直接压入堆栈进行操作。在 Dalvik 字节码中，本地变量会被分配到 2^{16} 个可用寄存器中的任意一个。Dalvik 操作码不会从堆栈中取值，而是直接从寄存器中取值并进行操作。
- （2）指令集：Dalvik 中有 218 个操作码，Dalvik 虚拟机中舍弃了在 Java 虚拟机中用于在堆栈和本地变量列表之间传递数据的操作码，Dalvik 虚拟机中的操作码更长，因为大多数指令都包含了操作所对应的源寄存器和目标寄存器地址。
- （3）常量池：JVM 字节码通常会遍历所有的.class 文件中的常量池。例如，保存方法名称的常量

池。在 Dalvik 虚拟机中,为了让一个常量池能够供所有的类使用,dx 舍弃了这种遍历的过程。而且,dx 还使用了“内联”的技术消除了一些常量。

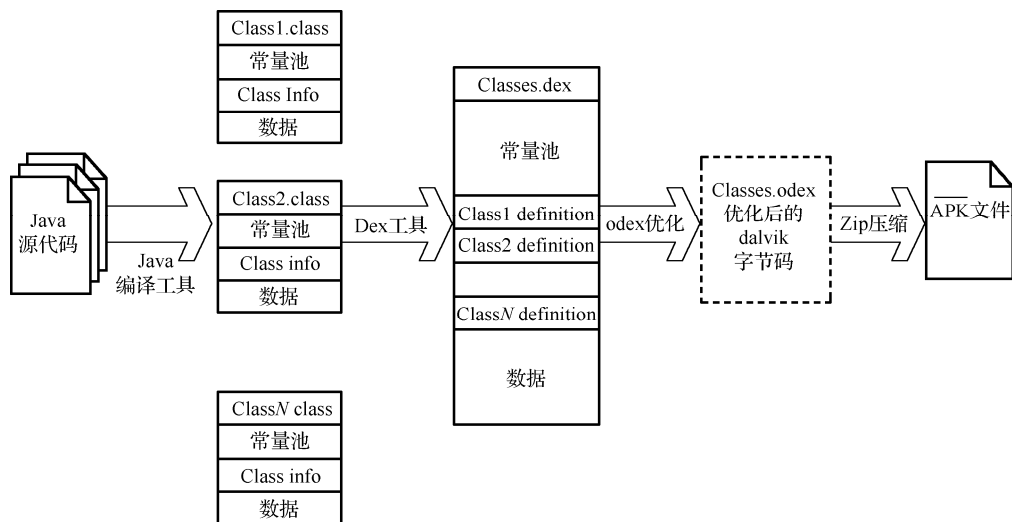


图 7-20 Android 应用程序生成过程

(4) 原始类型数组的存储: Dalvik 使用非确定的操作码来操作一个数组,与之相反的是, JVM 使用确定的操作码进行数组操作。因此,为了在翻译时能得到正确的数组类型, Dalvik 中的数组类型必须专门恢复。

(5) 空引用: Dalvik 字节码没有专门的 null 引用类型。取而代之的是, Dalvik 用常数 0 来表示 null。因此,对于常数 0 的含义往往需要进行适当的区分,从而避免引起歧义。

3. 垃圾回收机制

随着面向对象的编程语言应用越来越广泛,相比传统语言在堆上分配的数据所占的比例更高,数据结构和处理问题更为复杂,这让程序员管理内存更为复杂,程序员需要花费更多的精力在对象的分配与回收上,而自动内存管理技术可以有效提高软件的生产率。内存管理技术分为手动管理和自动管理,在 Java 语言中对象的内存管理是自动的,所谓自动管理是指对象分配的自动性和对象回收的自动性。在 Java 语言中对象的内存空间并不是在编译时就能够确定的,而是在 Java 虚拟机解释执行时才会为这个对象分配一段内存空间。与分配对象对应的过程就是回收对象过程,在 Java 中当对象不再被需要时需要释放所占用的内存空间,释放这段内存空间的过程同样是虚拟机自动完成的。

由于 Java 语言具有跨平台特性、安全性、自动内存分配管理技术、线程管理机制、动态类加载等特性,Android 选择了 Java 语言作为上层应用的开发语言。Java 语言在其内存管理上有别于 C/C++, Java 利用自动内存管理(即垃圾收集机制)跟踪正在使用的对象并发现清除不再使用的对象,这种方式将使程序员的工作难度和工作强度大大降低,也在很大程度上避免了内存泄漏问题的出现,但是垃圾收集的自动内存管理机制也给系统带来了性能方面的负面影响,比如内存消耗量大、性能下降、响应延迟、系统资源利用率下降、阻碍并发等。

Dalvik 虚拟机垃圾收集机制采用的是标记-清除垃圾回收算法, Google 自从 Android 2.3.1 开始对其做了并发的改进后,一直到 Android 4.4 仍然采用此算法,只是做了一些细节的改动,虽然 Google 从 Android 4.4 之后开始使用 ART 虚拟机,但是其中垃圾回收的算法仍然是标记-清除算法。在标记-清除回收算法中,应用程序在运行的过程中不停地创建新的对象直到内存用完,这时再创建新对象时,

系统暂停其他组件的运行，触发 GC 线程启动垃圾回收过程。内存回收的过程从根集开始，将整个 Java 堆遍历一次，保留所有可以被根集对象直接引用或者间接引用的对象，而剩下的对象就被当作垃圾并等待回收，伪代码如下所示。

```
VoidGC() {
    SuspendAllThread();           //挂起所有线程
    List<Object>roots = GetRoots(); //首先得到所有根集对象
    Foreach (Objectroot : roots) {
        Mark (root);              //将根集标记
        Process (root);
    }                             //标记根集直接或者间接引用的对象
    Sweep();                      //清除那些被标记的对象
    ResumeAllThread();            //恢复因为 GC 挂起的线程
}
```

算法通常主要分为标记阶段和清理阶段，在标记阶段主要涉及遍历对象引用过程，标记阶段的伪代码实现如下。

```
VoidMark (Object *pObj) {
    if (!pObj->IsMarked()) {
        pObj->Mark();
        List<Object *>fields = pObj->GetFields();
        Foreach (Object* field : fields) {
            Make (field);          //递归处理引用到的对象
        }
    }
}
```

清理阶段，执行垃圾回收过程，留下有用的对象。在这个阶段，GC 线程遍历整个内存，将所有没有被标记的对象全部回收，伪代码清单如下。

```
VoidSweep() {
    Object *pIter = GetHeapBegin();
    while (pIter<GetHeapEnd()) {
        if (!pIter->IsMark())
            free (pIter);          //如果没被标记就清理
        else
            pObj->Unmark();
        pIter = MoveNext (pIter);   //得到下一个对象
    }
}
```

标记-清除算法是比较早的垃圾回收算法，也是 Android 中垃圾回收算法的基础模型。标记-清除算法可以非常自然地处理循环引用，但是在垃圾收集运行时需要挂起系统的其他线程，影响系统的实时性，内存碎片严重，性能随着堆空间增加而下降。

在 Dalvik 的内存管理中，使用堆资源结构体管理内存中的各个堆空间，使用位图来进行标记对象是否存活，使用卡表来标记并发垃圾收集过程中对象是否改变。在 Dalvik 虚拟机中，Java 堆实际上是由一个 Active 堆和一个 Zygote 堆组成的，其中 Zygote 堆用来管理 Zygote 进程在启动过程中预加载和创建的各种对象，而 Active 堆是在 Zygote 进程 fork 第一个子进程之前创建的，之后无论是 Zygote 进

程还是其他子进程都在 Active 堆上进行对象分配和释放,这样做的目的是使得 Zygote 进程和其他子进程最大限度地共享 Zygote 堆所占用的内存。

对象的存活与是否需要标记位进行标记,简单的方法是在对象的头部增加标记位。这种方法不利于快速查找一个对象是否被标记,在标记-清除垃圾收集算法中,一种有效的组织方式是将标记位存放在一个分离的位图表中,在这个位图表中,各个二进制位关联堆中各个对象的起始地址。

位图中的“位”和 Java 堆中的对象是一一对应的关系,Java 堆的起始地址为 base,大小为 maxSize,创建的对象的地址范围为[base, maxSize]。并且 C 库提供的 mspace_malloc 在 Java 堆分配内存是按照 8 字节对齐的,这意味着我们只需要 $(\text{maxSize}/8)$ 个位(元素)来描述 Java 堆的对象。其中位指向的是一个类型为 unsignedlong 的数组,因此数组中的每一个元素可以描述 sizeof(unsignedlong) 个对象的存活情况。在 32 位的设备上,一个 unsignedlong 占用 32 位,这意味着需要一个大小为 $(\text{maxSize}/8/32)$ 的 unsignedlong 数组来描述 Java 堆对象的存活情况。如果换成字节数来描述的话,就是需要一块大小为 $(\text{maxSize}/8/32) \times 4$ 的内存块来描述一个大小为 maxSize 的 Java 堆对象。位图与 Java 堆的对应关系如图 7-21 所示。

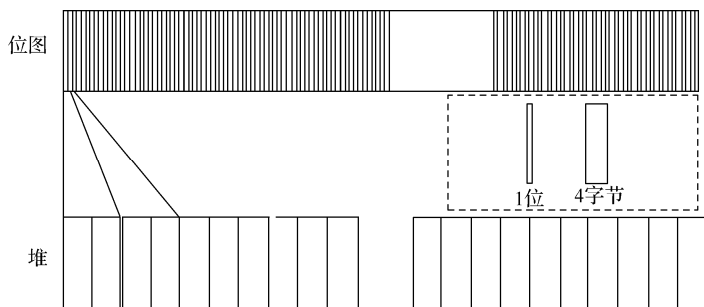


图 7-21 Java 堆中对象与位图的对应关系

4. JNI 技术简介

从编程语言来看,Android 系统由基于 Java 语言的应用层、应用程序框架层和基于 C/C++语言的系统运行库层、Linux 内核层组成,但是 Java 代码和 C/C++代码在实现原理和运行环境都具有很大的不同,并不能直接进行调用,必须提供一种媒介或者桥梁,将 Java 层与 C/C++层有机地联系起来。

JNI (Java Native Interface) 是 Java 层复用本地代码以及本地代码调用 Java 代码的一种技术,它是连接 Java 和本地代码的桥梁。JNI 技术主要的作用有两点:

- Java 程序中的函数可以调用 Native 语言编写的函数,Native 指的是 C/C++编写的函数。
- Native 程序中的函数可以调用 Java 层的函数,即 C/C++程序中可以调用 Java 的函数。

在执行 JNI 调用时,Java 层的每一个方法在本地层都有相应的函数实现,两者通过唯一的对应关系相关联。JNI 使用时一般考虑以下几种情况:

(1) 注重处理速度。与本地代码(C/C++)相比,Java 代码的执行速度要慢一些。如果对某段程序的执行速度有要求,可以使用 C/C++编写代码,之后在 Java 中通过 JNI 调用基于 C/C++编写的代码,提升运行速度。

(2) 硬件控制。硬件控制的代码一般位于硬件抽象层,使用 C 语言编写。通过 JNI 将硬件访问控制与 Java 层连接起来。

(3) 既有 C/C++代码复用。代码复用可以提高编程效率和保证程序的安全性与健壮性,Java 层代码复用 C/C++代码时,就要通过 JNI 来实现。

大多数程序位于上层的 Java 层，Android 通过把系统划分为几个层次从而使得开发者可以使用平台无关的 Java 语言进行 Android 应用开发，不必关心程序实际的硬件环境。Google 不仅为开发者提供了 SDK（Software Development Kit）开发套件，为了能让开发者使用 C/C++编写的本地化共享库，利用编译后的共享库更高效地完成计算密集型的操作来提高应用的性能，或者移植重用已有的 C/C++组件，提高开发效率。Android 1.5 之后又推出了 NDK（Native Development Kit）。在这种情形下，应用程序开发者通过 NDK 开发对性能要求较高的 C/C++模块再在 Java 程序中通过 JNI 将 Java 程序和 C/C++模块连接在一起，从而开发出运行速度快、执行效率高、健壮且安全的应用程序。

5. ART 虚拟机

在 Android 4.4 中，Google 开发了 ART 虚拟机来应对当今高性能嵌入式设备的需求。ART 虚拟机支持与 Dalvik 虚拟机相同的 Dex 文件，即应用程序开发者开发的原有 Android 应用程序可以直接在目标平台上运行，而 Dex 文件编译成目标平台本地代码的过程在目标平台上进行。

如图 7-22 所示，与 Dalvik 虚拟机类似，ART 虚拟机也包含类加载器、运行时数据区、垃圾回收器和本地方法接口等模块。作为一种新的 Android 虚拟机，ART 虚拟机采用了预编译系统取代 Dalvik 虚拟机的即时编译系统，执行引擎部分 ART 虚拟机中没有即时编译器，只包含快速解释器。在系统启动或应用程序安装时，ART 虚拟机调用 dex2oat 预编译工具，对应用程序安装包中的字节码进行验证与预编译，并将编译得到的本地机器代码，以及应用程序安装包中原有的 Dex 文件一起打包生成 ART 虚拟机运行时特有的 Oat 文件。应用程序运行时，ART 虚拟机直接执行 Oat 文件中的本地代码，运行效率大大提高。dex2oat 预编译工具并不会对所有的 Dalvik 字节码进行预编译，为了获得更高的执行效率，部分字节码通过 Dex 文件优化器（DexToDexCompiler）进行优化，并由快速解释器解释执行优化后的字节码。在创建 ART 虚拟机运行时的过程中，dex2oat 工具会将系统启动类路径中包含的系统 Java 库的 Dex 文件编译生成 boot.oat 和 boot.art 两个文件，前者是系统启动类路径中所有 Dex 文件编译生成的 elf 格式的 Oat 文件，后者是包含需要预加载的系统类对象的镜像文件，这两个文件生成后会被加载到运行时数据区。

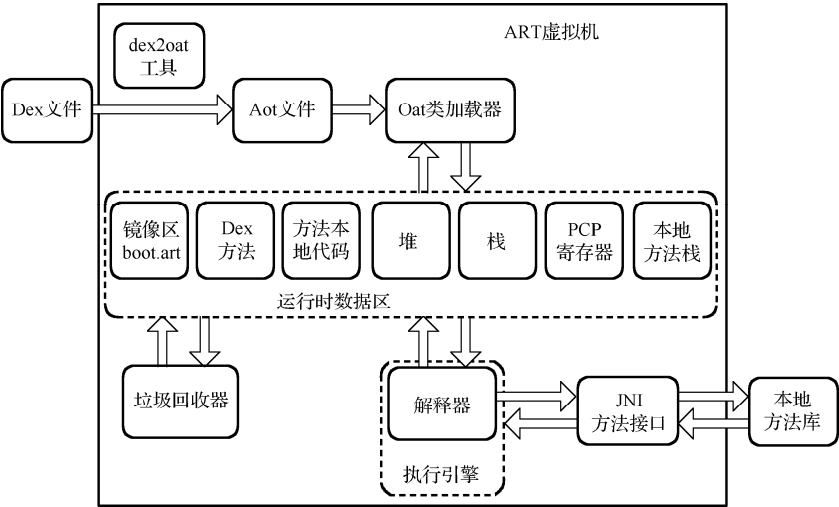


图 7-22 ART 虚拟机架构图

ART 虚拟机与 Dalvik 虚拟机相比最大的不同之处是用预编译器取代 Dalvik 虚拟机的即时编译器，将字节码的编译过程提前到应用程序安装时进行。与 Dalvik 虚拟机的即时编译器相比，ART 虚拟机预编译器具有许多优点。首先，Dalvik 虚拟机中，应用程序关闭后再次运行时需要重新对热点字节码段进行即时编译，重复编译会浪费系统资源，而 ART 虚拟机对字节码的预编译是在程序安装时一次性完

成的, 预编译生成的本地代码以 Oat 文件格式存储, 后续执行时直接执行 Oat 文件中的本地代码。其次, Dalvik 虚拟机的即时编译是在应用程序运行过程中进行的, 消耗系统资源, 对应用程序运行造成影响, 而 ART 虚拟机的预编译过程在程序运行前进行, 应用程序运行时直接执行预编译生成的本地代码, 效率更高。最后, Dalvik 虚拟机基于路径的即时编译器的编译对象是热点字节码段, 编译优化范围小, 而 ART 虚拟机的预编译器是针对整个应用程序, 具有更广的优化范围。

ART 虚拟机与 Dalvik 虚拟机一样采用标记-清除算法对内存进行回收, 并对垃圾回收器做了许多优化。ART 虚拟机在堆管理方面进行更加细致的划分基础上, 针对不同的使用场景, 设计了更加多样化的回收策略。不同的回收策略有不同的回收力度, 力度越大的回收策略, 每次回收的内存就越多, 消耗的时间也越长。为了降低垃圾回收过程对应用程序运行的影响, ART 虚拟机采用力度从小到大递增的垃圾回收策略, 如果小力度的垃圾回收完成后内存能够满足分配要求, 就不需要进行大力度的垃圾回收了。另外, ART 虚拟机充分利用处理器的多核特性, 在并行垃圾回收时将每个并发的垃圾回收任务划分给多个子线程, 从而可以更高效地完成垃圾回收。

6. LLVM 技术

Google 在 Android 系统中引入虚拟机技术的目的是为了更好地支持应用程序的跨平台性, 因为这也是 Java 语言的最大优势。为了方便应用程序员的开发, Android 还提供了便于使用的 SDK (Software Developing Kit, 软件开发包)。理论上, 所有基于 SDK 开发的应用软件都可以在不同硬件平台上无缝地迁移和运行, 实现一次编译、多处运行的理想。然而, 出于至少以下 3 个原因, 这个理想是非常难以实现的目标。第一, 由于 Java 本质上的解释执行机制 (虽然有 JIT 及时编译, 但这不改变该语言的解释执行本质), 使得与纯粹的编译语言相比, Java 程序的效率还是相对较低。ART 虚拟机的推出虽然将编译提前到安装应用程序之前, 在一定程度上缓解了效率的问题, 但其并未在根本上改变 Java 的基本架构。第二, 在 Android 之前已经有了大量基于非 Java 环境的软件模块, 这些模块大量采用 C 语言或 C++ 语言编写, 为了复用这些软件模块, 程序员必须将这些软件用 Java 语言重写, 显然这大大增加了软件开发成本。第三, 虽然 Java 源码可以编译成为二进制形式, 甚至对该二进制镜像进行加密, 但是存在大量的 Java 反编译工具可以将编译后的代码破解为可读性非常好的 Java 源码。这对于想保护软件知识产权的开发者而言几乎是一场灾难。

为了解决上述问题, Google 推出了 NDK 开发包, 通过该开发包提供的工具和框架, Java 程序员可以通过 JNI 接口调用采用 C 或 C++ 编写并已编译成为本地代码 (所谓本地代码是当前运行该代码处理器的机器码) 的二进制库。NDK 的引入似乎解决了上述 3 个问题, 程序员可以将效率敏感的函数采用 C 或 C++ 编写并编译成库, 这样可以大大提升系统的性能。事实上, 很多基于 Android 的游戏就是这么做的。另外, 由于 NDK 的引入也可以使程序员将现有代码迁移到 Android 平台的工作量大大降低, 因此可以把大量以前编写的 C 源码直接编译一下就可以了。同时, 由于反编译 C 或 C++ 的难度相对来说要大得多, 对于保护知识产权也有帮助。然而, NDK 的引入在很大程度上将 Java 最重要的优势丢了, 也就是跨平台性。比如, 某款针对 ARM 架构优化的 Android 3D 游戏可能就无法在 MIPS 架构的 Android 系统中运行。从这个角度上来说, 由于 NDK 的问题所造成的软件生态环境在很大程度上保护了 ARM 架构在移动终端应用领域的垄断地位, 这也成为国产嵌入式 CPU 所面临的最大困境。

能否有一种技术既能提供类似于 NDK 的性能优势, 同时又保留 Java 的跨平台优势呢? 为了解决这个问题, Google 公司在 Android 4.0 之后引入了底层虚拟机技术 (LLVM, Low Level Virtue Machine)。LLVM 项目始于 2000 年 12 月, 最初是由美国伊利诺伊大学研究开发的一个开源项目。与一般的编译器结构类似, LLVM 的基本结构由语言前端、中间优化器和后端目标代码生成 3 大部分组成。高级语

言（例如 C/C++语言等）处理前端将各种类型语言编写的源码转换到 LLVM 中间表示，由于 LLVM 框架的中间表示的语言无关性，中间优化器可以独立于前端和后端，对生成的中间表示进行各种优化工作。而进行中间优化后，必须通过后端目标代码生成器才能转换为具体目标硬件架构的汇编代码。LLVM 编译框架的基本架构如图 7-23 所示。

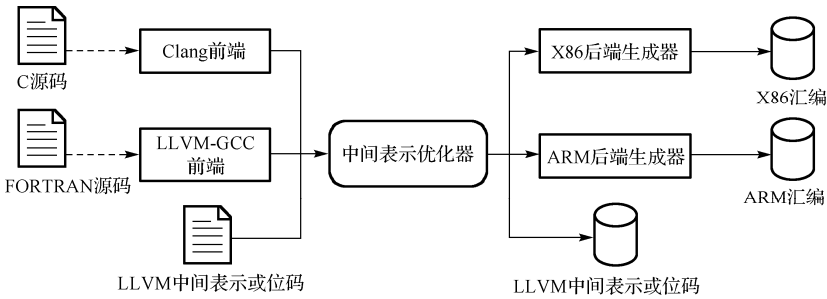


图 7-23 LLVM 编译框架的基本结构

LLVM 框架的后端处理过程的主要作用是将中间表示转换为目标指令，将 LLVM 架构的虚拟寄存器映射到目标平台的物理寄存器等。LLVM 框架后端由处理器无关的代码生成器和后端移植接口构成，改变后端移植接口就可以支持众多的处理器，目前为止，已支持 X86、ARM、MIPS、PowerPC、Hexagon 以及 Sparc 等处理器。这样模块化的设计提高了 LLVM 框架代码的可重用度，与此同时简化了对具体目标处理器进行后端移植的工作过程，提高了移植效率。

简单来说，用户编写的 C 或 C++代码在 LLVM 架构中并不是直接被编译为目标处理器的本地代码，而是被编译为 LLVM 的中间代码发布给用户，直到该代码被运行时才由 LLVM 的后端生成器动态地将中间代码映射到本地代码。由于中间代码与具体的硬件架构无关，因此可以最大限度地保持代码的跨平台性，同时又由于最终执行的代码是后端处理器动态编译的本地代码，因此其效率是可以保障的。Google 公司在 Android 4.0 之后引入的 RenderScript 编程框架就采用了 LLVM 技术，笔者认为目的之一就是为了解决 NDK 的移植性差的问题。

另一方面，随着嵌入式 GPU 的性能越来越强大，能否利用这些集成在 SoC 中的专用计算单元进行通用计算成为一个新的研究热点。利用 GPU 进行通用计算，学术界一般将其称为 GPGPU（General Purpose Computing on a GPU），为了支持 GPGPU，工业界和学术界目前比较流行的编程框架包括 Nvidia 公司推出的 Cuda 架构和开放的 OpenCL 架构。Google 还希望 RenderScript 架构能够成为 Android 平台的异构计算框架，因为理论上来说，LLVM 的运行环境只要配备相应的 GPU 编译器，就可以动态地将 RenderScript 程序（C99 标准的一个扩展）映射到 GPU 的操作指令。

7.3.3 Android 的任务间通信机制

Binder 是 Android 系统特有的进程间通信机制（IPC，Inter-Process Communication），其最初由 Be 公司在 1991 年开发的一款操作系统中所采用（OpenBinder），后经 Palm 公司继承开发所发展来的一套进程间通信机制。在 Android 系统中，Binder 机制进行了加强，其不仅仅是 IPC 机制，也是远程程序调用（RPC，Remote Procedure Call）机制。Android 的应用程序框架中，各种服务（Service）是核心，因此如何有效地调用服务很大程度上影响着系统性能。这其中，通过 Binder 调用的系统服务贯穿于 Android 系统的各个方面，Binder 的性能一定程度上制约着 Android 系统的性能。

1. Linux 的任务间通信机制

Android 系统基于 Linux 操作系统，Linux 已经拥有管道、System V IPC、套接字等 IPC 机制，Android

没有使用这些 IPC 机制，而使用了 Binder 来实现进程间通信。在进一步介绍 Binder 之前，有必要首先简单介绍一下 Linux 系统的原生通信机制。

在 Linux 和基于 Linux 的操作系统中，操作系统采用虚拟地址对用户进程进行管理。在 32 位操作系统中，用户进程的最大寻址空间为 4GB。每个进程都可以独享 0~3GB 的空间，这部分空间称作用户进程空间，而 3~4GB 之间的空间则留给内核使用，所有的内核空间是共享的。

正是因为上述原因，每个用户进程都拥有独立的虚拟地址空间，多个进程之间就存在着天然的隔离，一个用户进程无法通过访问另外一个进程的地址进行通信，这样就出现了各种进程间通信机制来实现进程之间的通信需求。

Linux 原生的进程间通信机制为 Linux 应用程序提供了标准化的数据通信模型，主要包括：

(1) 管道 (Pipe)。

管道分为无名管道和有名管道。

无名管道只能用于具有亲缘关系进程间的通信，允许一个进程和另一个与它有共同祖先的进程之间进行通信。一般情况下，无名管道由父进程创建，创建完成后管道会返回两个文件描述符，假设为 `fd[0]` 和 `fd[1]`，这两个文件描述符就代表读和写操作，然后由父进程执行 `fork()` 命令创建一个子进程，父进程和子进程分别开启读端或者写端，从而可以实现数据从父进程向子进程的传递。图 7-24 描述了从父进程到子进程的管道。

有名管道又称 FIFO，除具有无名管道所具有的功能外，还能用于无亲属关系的进程间的通信。有名管道在创建时需要指明管道路径，这样其他进程就可以根据这个路径对 FIFO 进行读写操作。

(2) 信号 (Signal)。

信号是一种异步通信机制，它与 Linux 的中断机制比较类似，用于通知接收进程有某种事件发生。除了用于进程间通信外，进程也可以发送信号给进程本身。在 Linux 中，信号常由系统函数发送，用来结束进程、设定定时器、进程阻塞等操作，这些信号都是 Linux 预先定义的。如果一个进程要处理某一种信号就需要在该进程中进行注册，即确定接收的信号和将要执行的操作之间的对应关系。信号并不是传统意义上的进程间通信机制，因为其不能对普通数据进行传输。

信号是一种异步通信机制，它与 Linux 的中断机制比较类似，用于通知接收进程有某种事件发生。除了用于进程间通信外，进程也可以发送信号给进程本身。在 Linux 中，信号常由系统函数发送，用来结束进程、设定定时器、进程阻塞等操作，这些信号都是 Linux 预先定义的。如果一个进程要处理某一种信号就需要在该进程中进行注册，即确定接收的信号和将要执行的操作之间的对应关系。信号并不是传统意义上的进程间通信机制，因为其不能对普通数据进行传输。

(3) 消息队列 (Message)。

消息队列是消息的链表，其表现与 FIFO 管道相似，但是它可以比管道承载更多的信息，管道只能传输无格式字节流，消息队列可以对传输的数据进行分配。具有写权限的进程可以向队列中增加消息，具有读权限的进程可以读取其中的消息。消息队列的发送进程不必等待消息接收即可返回完成其余的工作，同时接收方也不会一直等待消息到来，消息队列中的消息根据到达的先后顺序依次被接收。与 FIFO 还有一点不同的是，消息队列的消息存放总是放到队列最后，但是读取时却可以从中间读取。

(4) 共享内存 (Share Memory)。

共享内存是最有效率的进程间通信方式，管道和消息队列在进行进程间通信的时候都会经过两次复制操作，即复制到内核和从内核读取各进行一次。而使用共享内存不需要进行内核复制，共享内存通过对内核内存的映射，使得多个进程可以访问同一块内存空间。它往往需要与其他通信机制结合，如与信号量结合使用可实现进程间的同步与互斥。

(5) 信号量 (Semaphore)。

信号量主要作为一种同步手段，它实际上是在进程间进行资源访问时所用到的控制机制，本身并

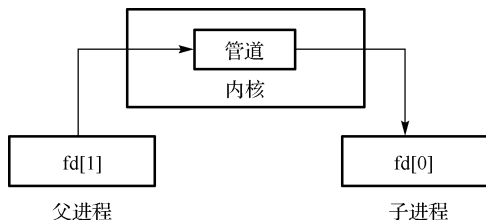


图 7-24 管道的数据传输

不进行有意义的数据通信传输。信号量相当于内存中的一个标志，进程可以根据这个标志来判断是否应该对某资源进行访问操作。

信号量是一种非负值的量，用于对临界资源进行访问管理。当信号量为正值时，进程可以使用该临界资源，并在访问临界资源的同时将信号量减 1；当信号量为 0 时，试图访问该临界资源的其他进程就会被内核强制进入等待状态直到信号量大于 0（这往往是因为占用该资源的进程释放了对该资源的占用），此时内核会将被置为等待的进程重新恢复为就绪态，并等待调度到运行态。信号量有一种特殊情况，它只有 0 和 1 两个值，称为互斥信号量，它常常被用于只能被一个进程访问的临界资源（这也是为什么称其为互斥的原因）。

（6）套接字（Socket）。

套接字是一种比较通用的进程间通信机制，既可以在本机内进行传输，也可以通过网络设备进行通信。所有提供了 TCP/IP 协议栈的操作系统几乎都提供了套接字，套接字对客户端和服务端的区分很明显，套接字具有 3 个属性：域、类型和协议。

套接字的域指的是在传输过程中所采用的介质，通常有 Internet 网络、UNIX 文件系统等。网络套接字的类型有两种——流套接字和数据报文。流套接字在 Internet 域通过 TCP 实现，由于有 TCP 协议的保证，流套接字在传输过程中数据能安全到达，而且还具有一定的纠错能力；数据报文使用 UDP 协议实现，传输效率高但是不可靠。文件系统套接字不利用网络进行传输，只在本地进行通信，它通过类 UNIX 系统的文件系统所开辟的通道进行传输，传输效率通常较网络套接字更高。

2. Binder 的 CS 架构

Android 是基于服务的操作系统，通过内置各种各样的服务来对应用程序提供系统最基本、最核心的功能，如设备控制、地理位置信息提供、定时设置等，这些系统服务都是通过 Android 的服务框架（Service Framework）实现的。

Service Framework 由两部分构成，一部分是使用 C++ 语言编写的本地服务框架，另一部分是使用 Java 语言编写的 Java 服务框架。图 7-25 描述了 Service Framework 在 Android 平台堆栈各部分相对应的情形。图中处于中间阴影矩形框内的部分即是“服务框架”，其中在矩形框的下半部分、位于本地库中的部分是 Native 服务框架（Native Service Framework），而矩形的上半部分、位于应用程序框架中的部分则是 Java 服务框架（Java Service Framework），它通过中间的 JNI 本地函数与本地服务框架进行交互。

应用程序框架层中的位置管理（Location Manager）、电话管理（Telephony Manager）、包管理（Package Manager）等 Java 系统服务都是使用 Java 服务框架实现的，而像图形图像渲染相关的 Surface Flinger 和音频相关的 Audio Flinger 这类本地系统服务都是用本地服务框架实现的。无论是 Native 服务框架还是 Java 服务框架，从图 7-25 所示的框架组成来看，都离不开 Binder 通信，可以说 Android 服务框架的核心就是 Binder 的通信框架。

Binder 是 Android 中所独特采用的进程间通信机制，它集合了其他 IPC 机制的优点并加以改进，以便更适合于 Android 应用程序，其主要特点为 Binder 既是 IPC 机制又是远程程序调用机制。Android 同时为 Java 环境和 C/C++ 环境提供了 Binder 机制，Binder 的最终实现主要依赖于它的 Binder 内核驱动。Binder 驱动提供了有关数据交换的一些功能，在此基础上，Android 实现了一整套基于 Binder 的进程间通信服务框架。

Binder 的设计架构使用了客户端-服务端（CS，Client-Server）架构，这种架构的好处在于可以支持异步通信，在使用大量通信时也可以保持高效运行；同时，它可以屏蔽 CS 架构的底层细节，使得开发者可以完全不用关心通信层的实现，对服务端的使用就如同运行在客户端中一样。Binder 的 CS 架构如图 7-26 所示。

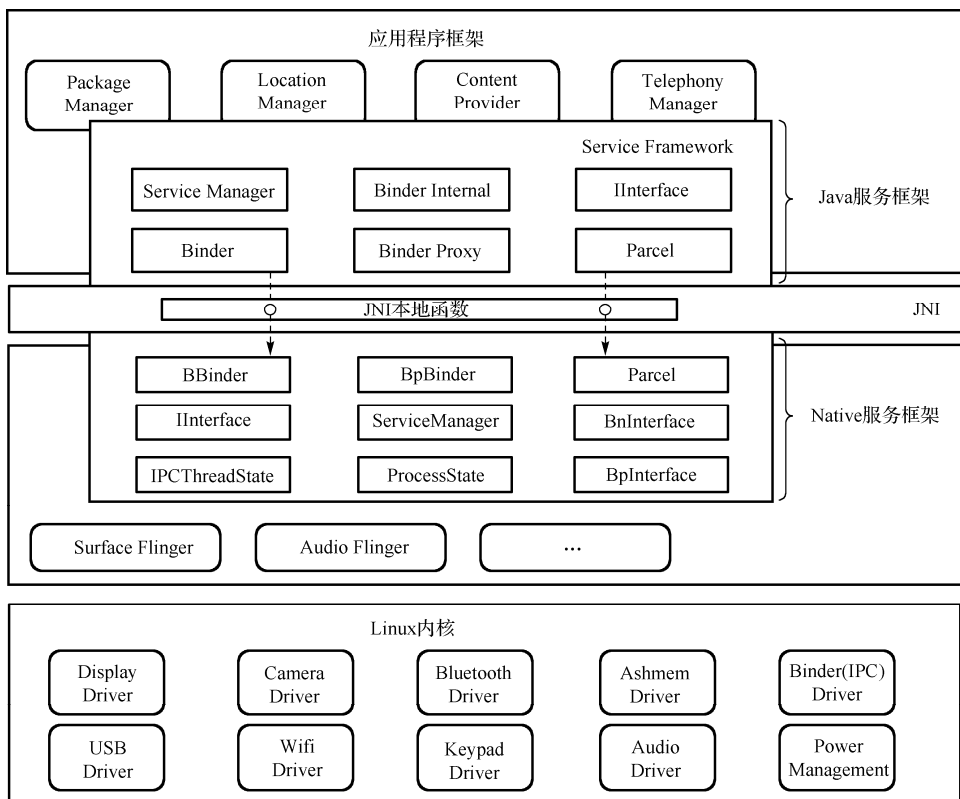


图 7-25 Android 的服务框架

从图 7-26 中可以看到，Binder 的 CS 架构由四大模块组成，其中通信的双方分别为客户端和服务端，都运行在用户空间，客户端要使用服务端提供的服务，就要通过 Binder 驱动和 Service Manager 来完成。Service Manager 是一个运行在用户空间的组件，负责管理和分配所有注册的服务。客户端在使用服务端提供的服务之前，需要在 Service Manager 中进行查询操作，由 Service Manager 返回一个服务端的代理对象，客户端才能进行真正的 IPC 通信。而所有的用户空间的注册、查询、使用操作都需要通过 Binder 驱动的参与来完成，Binder 驱动是一个虚拟设备驱动，用于管理地址的映射、Binder 对象节点的生命周期以及数据的复制传递的具体实现。

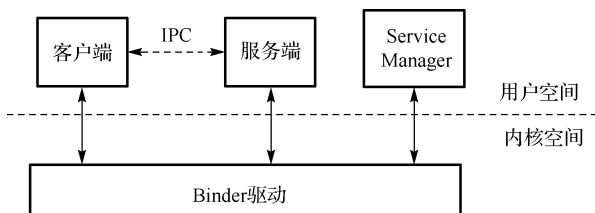


图 7-26 Binder 通信的 CS 架构

一次完整的 Binder 通信过程如下：

- (1) 服务端将自己提供的通信服务在 Service Manager 中进行注册，Service Manager 生成该服务的引用号，以供查询使用。
- (2) 客户端向 Service Manager 进行查询操作，获取服务的引用号。
- (3) 客户端利用获取的引用号在客户端进程中生成服务端的代理，并对代理进行操作，Binder 驱动通过代理端将数据传送到服务端去执行真正的操作，最后将传输结果返回到客户端。

3 个步骤中每次的交互都需要通过 Binder 驱动来完成，因此，深入分析 Binder 驱动程序的理解 Binder 数据传递的重点。Binder 驱动在 Android 中作为一个虚拟设备存在，运行在内核空间，设

备节点为“/dev/binder”。Binder 驱动所完成的工作主要有：

- (1) 管理所有使用 Binder 进程及其线程的状态保存和维护。
- (2) 内核空间缓冲区的映射与分配。
- (3) Binder 实体节点和引用节点的管理。
- (4) 完成最终的数据传递。

3. Binder 的通信模型分析

这里通过具体的位置管理服务 Location Manager 来分析 Binder 的通信模型。图 7-27 是用户程序对位置管理服务的获取及使用过程的 Binder 通信模型，客户端调用 LocationManger 的 getLastKnownLocation 方法获取最近一次的位置，其步骤如下。

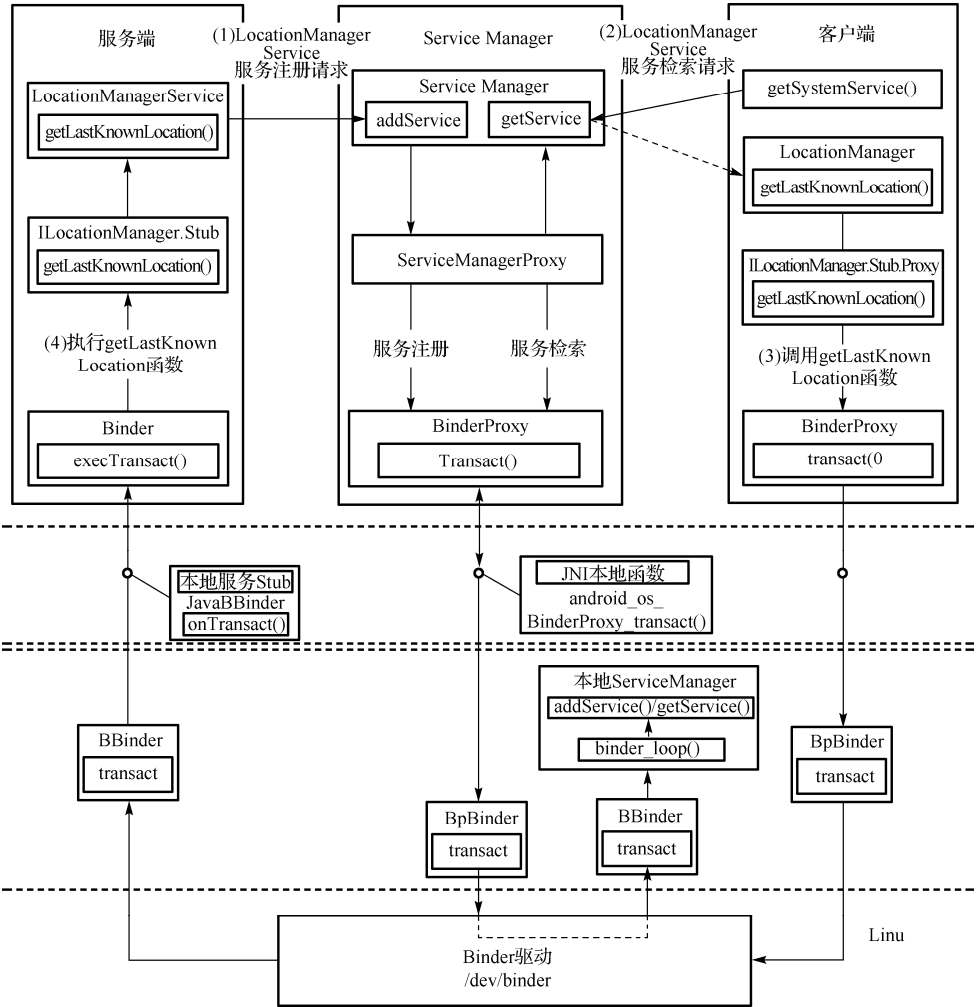


图 7-27 LocationManager 的 Binder 通信过程

(1) 服务注册: LocationManager 服务在注册自身服务时交互的对象是 Service Manager, 此时 Service Manager 是服务端, LocationManager 服务是 Service Manager 的“客户端”, 其注册过程本身也是一次完整的 Binder 通信过程。LocationManager 服务通过调用 ServiceManager 的 addService (Context.LOCATION_SERVICE, location) 方法注册服务。ServiceManagerProxy 服务代理将调用

addService()方法的信息转化为 RPC (Remote Procedure Call) 数据。Binder RPC 数据被存储在 Parcel 类中传递给 BinderProxy, 而后通过 JNI 传递给 BpBinder, 最后通过 Binder IPC 在 Service Manager 的 binder_loop()函数中将服务注册到系统中。

(2) 获取服务: 在使用 LocationManager 服务时, 客户端通过调用 SDK 中的 getSystemService(Context.LOCATION_SERVICE)来请求获取位置服务, 调用 getSystemService()方法将会引起 Service Manager 的 getService()方法的调用。若检索到 LocationManagerService 服务, 则会向客户端返回 LocationManager 对象, 该对象可以引用 ILocationManger.Stub.Proxy 服务代理。

(3) 调用 getLastKnownLocation()服务代理方法: 客户端获得一个 LocationManager 对象以后, 通过调用 LocationManager 的 getLastKnownLocation()方法获取最近一次的地理位置, 而后 ILocationManager.Stub.Proxy 将 getLastKnownLocation()方法的调用信息转化为 Binder RPC 数据, 通过 BinderProxy 经 JNI 传递给 BpBinder, BpBinder 通过向 RPC 代码中加入服务 Handle 信息, 最后经 IPThreadState 类添加 Binder 协议生成 IPC 数据后将其传递给 Binder Driver。

(4) IPThreadState 接收到 Binder Driver 传递 Binder IPC 数据并经过 executeComand()函数分析数据, 并将来自客户端的 RPC 代码与数据传递给 BBinder 类, BBinder 通过 JavaBBinder 调用 Binder 的 execTransact()方法。而后 RPC 数据被传递给 ILocationManager.Stub 服务的 onTransact()方法, 经过分析后调用 LocationManagerService 的 getLastKnownLocation()服务方法。

这样, 通过 Binder 进程间通信, 客户端通过获取一个 LocationManager 的服务代理, 并调用 getLastKnownLocation()服务代理方法, 便最终调用到了服务端 LocationManagerService 的 getLastKnownLocation()方法, 该方法会获取手机用户最近一次的地理位置信息, 并将执行结果经由 Binder 通信返回给客户端, 客户端也因此获取到了服务提供的地理位置数据。

从应用程序和位置管理服务的通信过程可以看出, Binder 整体通信原理与 Linux 系统调用类似。Binder 驱动像一个屏障, 将应用客户端和系统服务端隔离开来。并且, Binder 驱动限定了客户端必须按照既定的路线才可以调用到服务端对应的方法, 而方法本身属于系统服务程序, 真正的执行者是系统服务程序, 客户端由于 Binder 驱动的隔离无法直接调用使用, 它仅仅是等待服务端执行之后通过 Binder 驱动返回的结果。

7.3.4 Android 的安全机制

Android 系统是基于 Linux 内核开发的, Linux 是一个支持多用户、多任务的系统, 系统中文件的访问权限是通过用户 ID (UID) 和用户组 ID (GID) 来控制的, Android 系统不仅保留和继承了 Linux 内核提供的基于 UID 和 GID 的安全机制和进程隔离机制, 又在此基础上实现了一套自己的 Permission 安全机制。总体的 Android 系统安全框架如图 7-28 所示。

1. 继承自 Linux 的安全机制

1) 基于 UID 和 GID 的安全机制

用户和用户组的概念是 Linux 安全的核心。Linux 系统为每一个新创建的用户分配一个用户 ID (UID) 用于区分不同的用户。同时, 所有的用户又按组来划分, 每个用户组有一个组 ID (GID) 用于区分不同的用户组。同一用户可以属于多个用户组, 即一个 UID 可以对应多个 GID。

在 Linux 系统中, 权限以资源为单位进行分配, 这个资源通常是一个文件, 因为 Linux 将各种系统资源都视为文件进行统一处理。Linux 中的每一个文件都具有 3 种权限: 读、写和执行, 这 3 种权

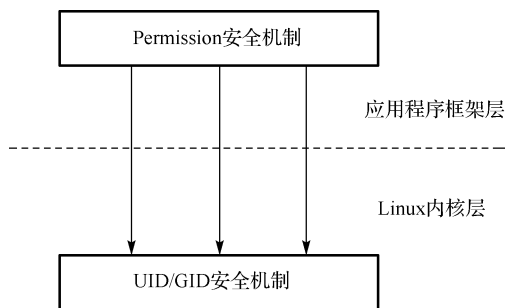


图 7-28 Android 系统的安全框架

限又按照用户属性划分为 3 组：文件所有者、同组用户和其他用户，这便是 Linux 的文件权限控制机制。由于每一个用户都可以对应多个用户组，因此，Linux 的每一个进程除了有一个对应的 UID 之外，还对应多个 GID，这些 UID 和 GID 就决定了一个进程所能访问的文件或者所能调用的系统函数，这就在内核级别上限制了程序的访问权限，保证了系统的安全。

2) 进程隔离机制

在 Linux 和基于 Linux 的操作系统中，操作系统采用虚拟内存管理机制对用户进程进行管理，以 32 位的 Android 操作系统为例，用户进程的最大寻址空间为 $0\sim 2^{32}\text{B}$ ，即 4GB 的寻址空间。每个进程都可以独享 $0\sim 3\text{GB}$ 的空间，我们称这部分空间为用户进程空间，而 $3\sim 4\text{GB}$ 之间的空间则留给内核使用，所有的内核空间是共享的。进程地址空间的布局如图 7-29 所示。

正是因为上述原因，每个用户进程都拥有独立的虚拟地址空间，这样多个进程之间就存在着天然的隔离，一个用户进程无法通过访问另外一个进程的地址进行通信，这使得任何一个用户进程由于执行错误指令或恶意代码导致的非法内存访问都不会意外改写其他进程的数据，一个用户进程的意外终止不会影响到其他进程的正常运行，从而保证整个系统的稳定性。

Android 在向下继承 Linux 内核提供的安全机制的基础上，还结合了移动终端的具体应用特点，进行了许多改进与提升，形成了 Android 的 Permission 安全机制。

2. Permission 安全机制

1) Android 的沙箱机制

Android “沙箱”的本质是为了实现不同应用程序之间的互相隔离。即在默认情况下，应用程序并没有权限访问系统资源或其他应用程序资源。Android 在安装程序时会创建一个新的用户，将新用户的 UID 分配给该应用程序，并且根据该应用程序的 AndroidManifest.xml 文件中所申请的权限将新创建的用户加入到对应的各个用户组中去。另外，Android 系统在/data/data/目录下会创建一个以程序安装包命名的新目录，该目录的所有者是为该应用程序创建的用户，应用程序创建的数据存储在该目录下。默认情况下，在该目录下创建文件时，文件的所有者是分配给该应用程序的 UID，对文件具有完全控制权，而文件所有者的同组用户和其他用户并没有设置访问权限，这样该应用程序就可以通过 Linux 的文件权限控制禁止其他应用程序访问自己的文件和有权访问其所属组的一些系统相关的文件了，Android 的沙箱机制在数据上实现了隔离控制。

除此之外，应用程序在运行时，系统会为每个应用程序创建一个虚拟机实例，每个应用程序在各自独立的虚拟机中运行，而每一个虚拟机实例实际上是一个 Linux 的独立进程空间，拥有独立的地址空间和资源，Linux 的进程隔离机制保证了不同的应用程序在不同的进程空间中运行，进程之间存在着天然的隔离，因此 Android 的沙箱隔离机制在空间上也实现了隔离控制。

2) Android 的权限控制机制

Android 的沙箱机制保证每个程序在资源受限的进程空间中运行，如果需要额外的权限或服务，则必须通过 Android 的权限控制机制去获得。Android 的权限控制机制主要用于限制应用程序对用户敏感数据、资源以及系统接口的访问。任何一个应用程序在使用 Android 受限资源（网络、电话、短信、蓝牙、通讯录、SD Card 等）之前都必须在 AndroidManifest.xml 文件中事先向 Android 系统提出申请。例如，一个读取用户联系人和访问网络的应用程序需要进行如下声明：

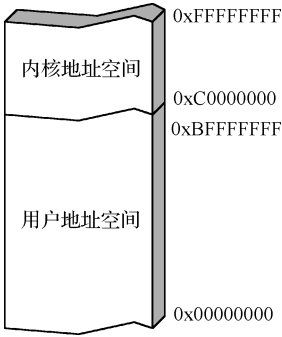


图 7-29 Linux 进程地址空间分布

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.app.ReadContacts" >
    <uses-permission android:name="android.permission.READ_CONTACTS"/>
    <uses-permission android:name="android.permission.INTERNET"/>
    :
</manifest>
```

当应用程序在安装时,系统会读取 `AndroidManifest.xml` 文件中的权限并申请相应的权限,Android 系统提供的相关权限都位于 `frameworks/base/core/res/AndroidManifest.xml` 中。

Android 的权限控制主要分为两类,一类是对设备的直接访问权限,另一类是对设备或者服务的间接访问权限。

对设备的直接访问权限是指访问网络、蓝牙等设备时所涉及的权限,Android 系统直接使用了 Linux 内核提供的文件权限控制机制,采用用户组的形式管理该类型的权限。Android 系统在 `frameworks/base/data/etc/platform.xml` 文件中记录了用户组 GID 与权限字符串之间的对应关系。例如,对于访问网络的权限其对应的 GID 如下:

```
<permission name="android.permission.INTERNET" >
    <group gid="inet" />
</permission>
```

这样申请访问网络权限的应用程序在安装时,除了获得自身的 UID 和 GID 以外,还被添加到相应的 `inet` 用户组中,也就因此获得访问网络的权限,Permission 机制中的权限控制机制也因此和 Linux 文件访问控制机制联系到了一起。

对设备或服务的间接访问权限是指读取手机联系人数据、短信数据、手机设备信息等相关的权限,这种访问是通过 Android 的 Service 服务机制来实现的。Android 系统是基于服务的系统,通过采用 Client-Server 的方式为运行在 Android 系统上的应用程序提供服务。Android 系统中内置了各种服务用于管理系统受限的资源,如用于手机通话相关的电话管理服务 `Telephony Manager`、用于管理手机位置信息及基站信息的位置管理服务 `Location Manager`、用于应用程序提供其他应用程序访问(如手机联系人数据、短信息数据的内容提供器 `Content Provider`)等。当一个应用程序在运行过程中需要访问系统提供的某个功能时,会向提供该功能的服务端发出请求,服务端在收到请求以后,会检查该应用程序是否在安装时申请了相应的权限。如果应用程序没有权限,服务端拒绝为应用程序提供服务。服务端的这种运行时权限检查是通过 Android 特有的 `Binder` 进程间通信机制在包管理服务(`PackageManager Service`)中获取检查结果的,应用程序通过安装时向系统申请权限来获取服务端的信任。

图 7-30 显示了申请访问系统手机联系人的应用程序 `com.test.getContacts` 与提供联系人资源的服务程序 `com.android.provider.contacts` 的沙箱的运行及通信情况。图中,包名为 `com.test.getContacts` 的应用程序 A 和包名为 `com.android.provider.contacts` 的服务程序 B 分别运行在各自的沙箱中。应用程序 A 在自己的 `AndroidManifest.xml` 中申请读取用户联系人的权限 `<uses-permission android:name="android.permission.READ_CONTACTS"/>`,服务程序 B 提供了访问联系人数据的服务。当程序 A 想获取用户联系人数据时,便通过进程间通信机制 `Binder` 访问了提供手机联系人服务的程序 B,服务 B 收到访问请求后会调用 `enforceReadPermission` 方法通过 `Binder` 通信机制获取 `PackageManager Service` 提供的权限检查结果,之后根据该结果决定是否返回手机联系人数据给程序 A 或者拒绝访问。

3) Android 的应用程序签名机制

签名机制在 Android 应用框架中也具有非常重要的作用,无论是系统应用程序还是普通应用程序都必须由开发者持有的证书来进行签名。Android 签名机制主要有 3 个作用:

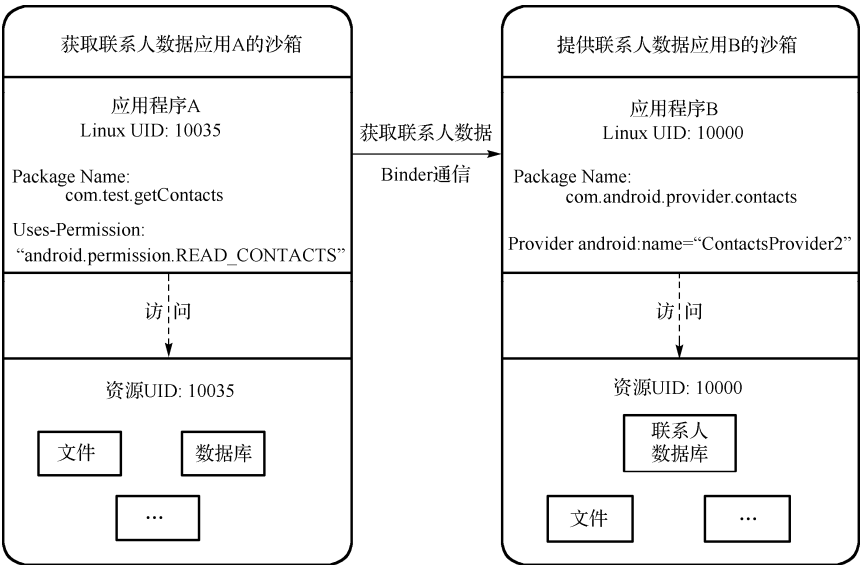


图 7-30 Android 沙箱获取联系人受限资源

（1）对程序包的完整性和可靠性认证。签名机制可以防止病毒程序植入正常应用程序来篡改程序包的内容，保证了程序包的完整性和可靠性。

（2）有利于应用程序升级。应用程序进行版本更新时，必须使用同一个证书进行签名，这是由于只有使用相同的证书，系统才会认为它们是同一程序的不同版本，允许新版本的程序安装。

（3）有利于代码或者数据共享。对于拥有相同数字证书的应用程序可以共享功能或者数据。

3. Android 平台的安全性挑战

1) Android 中的用户隐私信息

在 Android 手机中，保存了用户的短信、联系人信息以及通话记录等各类用户数据。在 2014 年 9 月，中国互联网数据中心（DCCI，Data Center of China Internet）联合 360 手机安全中心发布了《2014 年上半年 Android 手机隐私安全报告》。报告中显示，智能手机已成为用户隐私信息最多的载体，并且 Android 手机中用户隐私数据的安全存在很大的威胁。DCCI 认为 Android 中和隐私相关的权限包括访问联系人、获取设备信息、读取短信记录、获取通话记录、读取位置信息等 12 项权限。通过对 1200 个 Android APP 测试发现，92%的应用获取了隐私权限。

而日常生活中恶意软件收集的隐私信息主要包括用户的短信记录、手机通讯录、通话记录、位置信息等，并通过网络连接在用户毫不知情的情况下发送出去。

针对在 Android 手机中保存的数据信息以及恶意软件主要窃取的用户信息，笔者认为在 Android 手机平台中，用户的隐私信息包括了手机中保存的短信信息、联系人信息、通话记录、日程安排及 Android 手机标识 IMEI 号，以及通过 Android 手机获得的用户位置信息这六大类信息。

2) 安全机制的缺陷

虽然 Android 系统通过继承 Linux 内核的安全机制和自身的 Permission 机制实现了对系统资源和敏感数据的安全保护。但是，面对错综复杂的使用环境，用户隐私信息仍然面临着巨大的威胁，主要表现在以下几方面。

（1）粗粒度权限控制。

应用程序所需的权限必须通过用户授权才能使用，授权提示界面会在应用程序安装时显示，在授权提示界面中，用户只能要么同意所有申请的权限或者要么全部否决，这是一种“All or Nothing”的

权限模型。这种粗粒度的权限控制机制使得用户为了使用某些应用程序而不得不接受该应用程序的所有权限申请，这就有可能造成用户隐私数据的泄露。并且，当应用程序安装后，授予的权限不能动态修改，即使用户知道某些与隐私信息相关的权限有数据泄露的危险也无法改变，除非卸载该应用程序。

(2) Android 权限机制不够透明化。

虽然越来越多的用户意识到隐私数据的重要性，但用户对于 Android 中相关权限的认识程度及权限和隐私数据的关系了解得并不够清楚。因为在目前 Android 平台的安全机制中，当应用程序使用已经申请的权限去获取服务时，系统不会再弹出相关通知提示用户，用户因此对应用程序申请的相关权限在后续过程中的使用情况也根本无法得知。如何让用户透明地认识应用程序相关权限的后续使用情况，是提升用户对 Android 权限理解的关键，也是用户在 Android 手机上保护自身隐私数据的关键。

案例：基于 SEP4020 的 EPOS 软件平台设计

EPOS 是在普通通信终端的基础上，开发支持金融安全加密的 POS (Point of Sale) 支付功能，并兼容简单的数据通信业务，即固网或者移动支付终端。由于具有安全、简便、稳定等特点，这一新型刷卡方式受到越来越多用户的青睐。目前国内主流厂家把这种支付终端称为 EPOS。简单来讲，EPOS 支付就是中国电信、网通、联通等电信运营商和中国银联联手推出的一种基于现有通信网络进行电子支付的创新支付终端，该终端把银行卡的支付服务延伸到了消费者身边。只要用户家中有直拨电话线，那么只需更换一部可刷卡的电话即可进行刷卡消费了，而且这样一部 EPOS 智能刷卡终端的价格只有传统意义上 POS 终端的 30% 左右。

1. EPOS 系统对金融交互式终端产品的要求

EPOS 系统对金融交互式终端产品的要求如下：

(1) 考虑到系统开发平台的统一性以及对于后续在线升级的需要，越来越多的系统设计者选择嵌入式 Linux 作为操作系统。

(2) NandFlash 以其低成本的特点已经越来越受到系统设计者的欢迎。另外，芯片支持直接从 NandFlash 启动可以进一步降低系统设计的成本。

(3) 由于金融交互式终端往往对于系统的安全性具有较高要求，因此一般需要通过 IC 卡进行数据的加密，这就需要处理器能够支持符合 ISO7816 标准的 IC 卡控制器。

(4) 金融交互式终端一般需要外接热敏打印机或者针式打印机进行票据打印，而对于热敏打印机和针式打印机的支持需要较多的硬件定时器和通用 IO 口。

(5) 随着互联网技术的发展，越来越多的金融交互式终端需要通过以太网接口与互联网相连，因此内嵌以太网控制器的处理器将大大降低系统设计成本。

(6) 金融交互式终端一般需要通过多个串口与其他设备进行数据传输，比如通过串口控制 GPRS/CDMA 通信模块、通过串口控制银联直连 Modem、通过串口连接红外条码扫描器等。

(7) 随着彩色 TFT LCD 显示屏的价格越来越低（目前，彩色 LCD 的价格甚至比同分辨率的黑白 LCD 更加便宜），越来越多的金融交互式终端需要配置彩色 LCD 显示屏，通过在芯片内部实现 LCD 控制器可以进一步降低系统设计成本。

目前 EPOS 终端设计方案中采用的主要处理器为传统的 x86 处理器、8 位/16 位单片机，以及以 ARM 处理器为代表的嵌入式微处理器。通过调查了解到，由于传统意义上的 x86 的工控处理器架构成本居高不下，并且技术门槛高、功耗和体积大、不易集成等缺点以渐渐被以 ARM 架构为代表的 32 位处理器所替代，慢慢退出了嵌入式系统领域；而成本低、技术难度小的单片机由于其处理能力、扩展能力的局限性，已经远远不能满足目前终端市场的需求，只能针对一些功能、结构要求简单的低端市

场开发特定的应用，随着嵌入式 32 处理器的成熟和量产化，成本不断降低，其最终将成为嵌入式领域的主力军。目前市场上主流金融交互式终端绝大多数也是采用的 ARM 内核的嵌入式微处理器。

尽管目前客户在终端方案中采用了 32 位的 ARM 嵌入式微处理器，如三星的 S3C44B0、S3C2410，NXP 的 LPC21XX、LPC22XX 微处理器等，但这些处理器多是专门针对工业控制类或手持 PDA 类应用，处理器内部缺乏针对金融交互式应用的功能模块，如没有智能卡控制器、缺少以太网控制器、标准 UART 数量过少、硬件定时器数量不足。这些功能模块的缺失只能通过外扩芯片实现，带来的直接后果就是系统复杂度提高、终端成本上升。另外，由于市场上绝大多数方案采用的是国外公司的处理器，这些公司在国内的技术力量薄弱，支持力度不足，也导致终端厂商开发难度加大，开发风险高，成本上升。

SEP4020 微处理器是一款面向金融类交互式终端的 32 位嵌入式 ARM 微处理器，采用高性能的 ARM720T 内核，设计有丰富的外围功能模块以满足此类应用需求。SEP4020 的功能设计充分考虑了金融类交互式终端应用，其主要特点包括：

(1) 处理器内核采用 ARM720T，内含标准的 MMU 和 8K Cache，支持标准的嵌入式 Linux 操作系统。一方面由于嵌入式 Linux 是开源的操作系统，产品开发前期无需购买相关 Licence，降低了系统开发成本和开发风险；另一方面，由于 Linux 系统是一个开放的系统，拥有众多的开放软件资源，从而加速了产品研发周期。

(2) SEP4020 内嵌了 NandFlash 控制器，并支持直接从 NandFlash 启动，降低了系统中非挥发存储器的成本。

(3) SEP4020 中的 SMC（Smart Card Controler，智能卡控制器）就是为金融类应用定制的，CPU 直接支持 ISO7816 的智能卡，特别是金融类行业常用的 PSAM 卡、税控 IC 卡，为客户节约了外扩芯片的成本并降低了开发难度和系统复杂度。

(4) 高速的 PWM/GPIO 可以直接模拟热敏打印的串行数据，最高速度可达 10Mbps 以上，外围只需加少量的驱动电路即可驱动串行的高速外设，提供了灵活多变的设计方案。

(5) SEP4020 内嵌 10M/100M 以太网控制器，系统设计无需外挂以太网控制芯片，降低了系统成本。

(6) 四个标准 UART（其中一个支持硬件流控）方便用户在进行产品设计时与其他设备的数据交换。

(7) SEP4020 内嵌的彩色 TFT 液晶控制器，可以方便地与彩色 LCD 相连，支持彩色显示终端。

(8) 除了上述各个功能模块外，SEP4020 还集成了 USB 总线、SD 卡控制器、IIS 音频接口等，大大方便了用户进行系统扩展时的设计需求实现。

2. EPOS 软件平台框架

一般而言，嵌入式软件的体系结构包括 BootLoader、操作系统层和应用层，如图 7-31 所示。

(1) BootLoader/BSP：由于硬件平台的差异性，需要根据硬件的具体情况编写系统初始化代码。一般而言，BootLoader 完成必需的硬件初始化，并初始化操作系统运行需要的软件环境，最终引导操作系统运行。

(2) 嵌入式操作系统：操作系统是嵌入式系统的一个重要组成部分，由于操作系统的加入，系统设计难度将大大减少，也缩短了系统开发周期。操作系统通过设备驱动管理底层硬件，并通过 API 的形式向应用程序提供接口。

(3) 应用层软件：嵌入式系统的应用层软件实现了系统的功能。嵌入式应用软件是针对某个特定的应用领域，基于固定的硬件平台，来达到用户目标的计算机软件。一般而言，把图形用户界面（GUI）、文件系统以及协议栈等软件中间件也视为应用层软件的一部分。

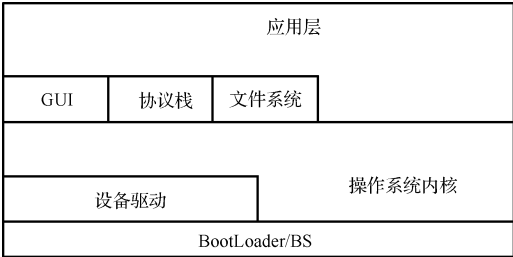


图 7-31 嵌入式软件体系架构

EPOS 软件平台的设计采用 U-BOOT1.1.4 作为系统的 BootLoader，并采用了 Linux2.6.16 操作系统，在应用层采用 MiniGUI 1.3.3 开发图形用户界面，软件系统架构如图 7-32 所示。

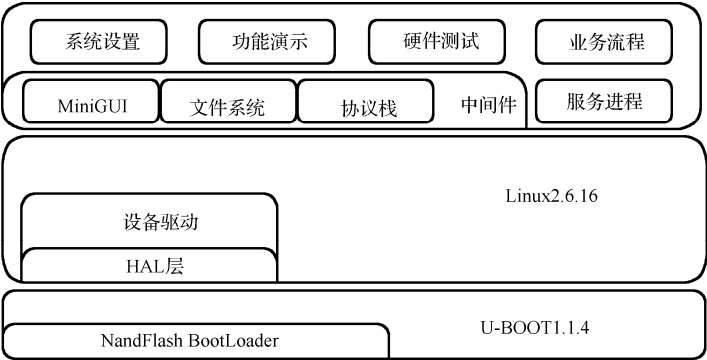


图 7-32 EPOS 软件系统架构

EPOS 终端平台基于 32 位 ARM720T 内核微处理器 SEP4020，具有高性能、低成本、扩展性强的特点。该平台不仅提供用户一套基于 32 位微处理器芯片的硬件平台，还包括整套的硬件方案、底层驱动代码、操作系统以及参考应用设计。采用了高性能 32 位 RISC 处理器和大容量存储器，支持磁条卡、IC 卡、PSAM 卡、外扩 EMV 卡等多种交易卡；处理器外部接口灵活，可以采用基于电话线 FSK/DTMF 传输方式，可以基于 GPRS/CDMA1X 的无线接入方式，也可以采用基于以太网的接入；在协议上支持银联网控制器接入要求，内嵌银联 ISO8583 直连 POS 协议；内部支持针式打印方式，外部扩展支持加密键盘和扫描枪；该平台可以通过终端机实时的交互菜单、迅速及时的信息公告与智能快捷键的设置，配合交易中心实现整个电子交易系统的智能化电子交易过程。

表 7-3 UB4020_POS_EDU 开发平台功能列表

部件		功能特点
核心部件	CPU	<ul style="list-style-type: none">● 具有 16 位数据总线和 23 位地址总线的微处理器● RISC 内核的频率为 96MHz● 64K 字节的嵌入式 sram
	Timer/Counter	6 个 32 位 timers/counter 和 4 个 16 位 timers/counter
	I/O Port	最多支持 97 个 I/O
	Nor Flash ROM	基本储存空间 2MB，可扩展至 8MB
	NandFlash ROM	基本储存空间 64MB，可扩展至 4GB
	SDRAM	基本存储空间 32MB，可扩展至 64MB
	UART	内置标准串行异步（RS232）通信口
	IrDA	内置标准红外（IrDA）通信口
	WatchDog	处理器自带看门狗，保障系统不间断运行
	工作电压	CPU 工作电压，Core：1.8V，I/O：3.3V
	休眠	系统运行速度有多种模式，具有休眠等低功耗功能
	键盘	支持数字/英文/汉字输入
输入	PSAM 卡	支持 2 个 PSAM 卡座
	外扩 EMV 卡	支持两通道外扩 EMV 卡，兼容 EMV3.1.1 和 EMV4.0
	磁卡阅读	刷卡可读符合 ISO2/3 标准的磁卡
	IC 卡读写	可读取符合 ISO7816 标准的 CPU 卡以及普通存储卡
输出	打印机	针式打印机，自带汉字库，可打印汉字、点阵图形 热敏打印机，自带汉字库，可打印汉字、点阵图形
	LCD	处理器内置液晶控制器，支持 STN 黑白液晶显示屏和 TFT 彩色显示屏，标准配置：320×240（含触摸），最大分辨率可至 800×600
	信息灯	有键盘灯、工作灯、LCD 背光灯等功能

续表

部件		功能特点
通信	以太网接口	支持 TCP/IP、PPPOE 等协议，支持基于 IP 的交易服务
	GPRS 模块	支持无线 GPRS 通信及短消息收发
	CDMA1X 模块	支持无线 CDMA1X 通信和短消息收发
	DTMF/FSK	电信 DTMF/FSK 信号产生器
	MODEM 模块	符合银联接入规范

该平台整合了 EPOS 等交互式设备所需要的几乎所有的相关设备，为终端用户的产品开发提供了样例设计。

EPOS 平台涉及的硬件驱动繁多，既包括 SEP4020 相关的 NandFlash 驱动、RTC 驱动、USB Device 驱动等 CPU 内部模块驱动，也包括磁条卡、热敏打印机、FSK 通信等外扩模块驱动。

思考题

- 1. 如果一个操作系统中采用 Idle 任务作为最低优先级任务，为什么该任务必须永远处于就绪态？如何才能保证该任务总是处于就绪态？（提示：在编写 Idle 任务代码时，可以在该任务中调用所有的操作系统的系统调用吗？）
- 2. 如果对于不可剥夺型内核而言，调度只发生在改变任务状态的相关系统调用中，对于抢占式内核（也就是可剥夺型内核）而言，调度除了发生在上述系统调用中还会发生在哪些地方？
- 3. 为什么全局变量是临界资源，而局部变量（采用 static 修饰的局部变量除外）不是？
- 4. 嵌入式操作系统中的 system tick 也称为系统滴答，也称为一个时间片，每当时间片到期后，操作系统会检查是否有其他优先级更高的任务等待运行，那么这个 system tick 是如何生成的呢？

扩展阅读

[1] 塔嫩鲍姆. 操作系统设计与实现（第三版）（上册）[M]. 电子工业出版社，2015.

[2] Lee, Edward A. What's ahead for embedded software?[J]. Computer 33, no. 9 (2000): 18-26.

[3] 韩超，梁泉. Android 系统原理及开发要点详解 [M]. 电子工业出版社，2010. 113-115.

[4] 凌明. 嵌入式系统高级 C 语言编程[M]. 北京航空航天大学出版社，2011.

[5] 方海贞. RenderScript 异构计算框架的应用与优化 [D]. 东南大学硕士论文，2015.

[6] 李霞. Android 虚拟机运行时技术的分析与评测 [D]. 东南大学硕士论文，2015.

[7] 甄鑫. 基于 Binder 的 Android 用户隐私数据安全增强技术实现[D]. 东南大学硕士论文，2015.

[8] 苑冰泉. Android 进程间通信机制 Binder 的分析与对比研究 [D]. 东南大学硕士论文，2014.

[9] Chin, Erika, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android[C]. In Proceedings of the 9th international conference on Mobile systems, applications, and services, pp. 239-252. ACM, 2011.

[10] 高晓东. Dalvik 虚拟机指令扩展与优化 [D]. 东南大学硕士论文，2014.

[11] 黄洋. Android 内存管理中的垃圾收集性能优化 [D]. 东南大学硕士论文，2014.

[12] Shabtai, A., Y. Fledel, U. Kanonov, Y. Elovici, and S. Dolev. "Google Android: A State-of-the-Art Review of Security Mechanisms." arXiv preprint arXiv:0912.5101 (2009).

[13] Cheng, Lu. "Analysis and Comparison with Android and iPhone Operating System." University of Central Florida Department of Electrical Engineering & Computer Science. Retrieved October 6 (2011).

第8章 嵌入式系统功耗优化

8.1 嵌入式系统功耗优化概述

8.1.1 嵌入式系统的功耗问题

嵌入式系统是多个设备或者对象的组合，其在一定的限制条件下相互作用可产生特定的功能。现在已经出现了许多测试标准来对嵌入式系统的整体设计质量进行评估，如系统性能、稳定性、能耗、设计和生产费用等，其中系统的能耗与功耗问题在最近几年已经逐渐成为一个重要的设计考虑因素，其重要性随着手持设备（通常采用电池供电）的普及而越来越突出。在以往的设计过程中，系统的稳定性、实时性、安全性等是设计和考虑的重点，但是现在对系统设计者来说又产生了一个新的挑战——降低系统的能耗，其必要性体现在以下几个方面：

（1）相当多的嵌入式设备，尤其是手持与便携式设备采用电池供电，而电池容量相对有限，因此需要通过降低设备功耗延长一次充电后的设备使用时间，提高产品竞争力。

（2）半导体工业的迅速发展使得集成电路（IC，Integrated circuit）的集成度和时钟频率得到了显著提高，但运算能力爆发性增加的同时也导致了系统的功耗急剧上升，这将带来散热的问题，也增加了设备的封装费用。

（3）电池技术的发展速度严重滞后于系统能耗的增长速度。在最近30年中电池容量只增长了4~8倍，而数字IC运算能力的增长却超过了4个数量级。

（4）能量价格的上浮、绿色节能理念的深入人心以及人们对电磁辐射等环境问题的关注程度越来越高，也进一步表明了降低系统功耗的重要性。

嵌入式系统是软硬件协同设计的过程，其系统组成和应用特性提供了低功耗策略设计的可能性，这些可能性包括设备/芯片多种工作模式、工作负载多变和系统嵌入标准化3个方面：

（1）设备/芯片多种工作模式：嵌入式系统中，除了正常功耗模式外，越来越多的设备/芯片还支持一种或多种低功耗模式，这就为功耗管理提供了可能，即系统可根据工作负载的变化来设定设备/芯片的工作模式。此外，SoC芯片低功耗技术包括电源门控、动态电压调节技术使得系统级的动态电压调节和动态功耗管理成为可能。

（2）工作负载多变：嵌入式系统是多种不同特性的器件的结合，系统通常为最坏情况下达到峰值性能而设计，但通常处于欠负载工作状态，且工作负载具有不均匀性。这种工作负载的多变性为系统能量的自适应降低提供了可能性。

（3）系统嵌入标准化：当设计出一个具有节能效果的动态低功耗策略后，还需将其嵌入到整个系统程序中才能实际应用。目前主流的操作系统（例如Windows和Linux）都支持高级电源管理（APM，Advance Power Mangement）、高级配置和电源接口（ACPI，Advanced Configuration and Power Interface）等模块。APM是由Intel和微软联合开发的一组API（Application Programming Interface，应用程序接口），目的是允许操作系统和BIOS协同进行电源管理。ACPI被Intel、Microsoft、Toshiba推荐为系统功耗管理中的标准硬件接口，允许设备厂家、操作系统设计者和设备驱动编程人员使用同一个标准接口。通过引用或改进APM、ACPI接口可以很容易地将低功耗设计策略嵌入到系统内核中。

由于嵌入式系统的工作负载通常会随着时间发生变化，那么通过关闭设备（动态功耗管理，DPM）或动态调节处理器的工作电压（动态电压调节，DVS），就能取得系统性能和功耗之间的平衡，从而降低嵌入式系统的功耗。

能量的高效使用除了能够降低系统操作代价（例如电能消耗）和减小环境影响（例如辐射干扰、噪音）外，对于手持设备而言，更重要的是延长了使用时间与待机时间。为了在系统性能得到保证的同时降低系统能耗，需要同时对硬件和软件进行设计优化。对硬件电路来说，随着集成电路工艺尺寸的持续减小，芯片面临的散热问题越发严重，降低功耗在近年来一直是紧迫的任务。但对于整个嵌入式系统来说，不能仅从功耗角度来看。

在继续我们的讨论前，首先需要明确什么是功耗和能耗。

能耗和功耗是用来衡量系统消耗能量的两个重要概念，在不同的上下文中，能耗和功耗的定义不同。在计算机系统中，能耗是计算机系统一段时间内总的能量消耗，单位是焦耳（J），其定义如下：

$$E = \int_t^{t+\Delta t} P \cdot dt$$

其中， E 表示系统的总能量消耗，由时间 t 和功耗 P 两方面因素共同决定。功耗是指单位时间内能量的消耗，反映计算机系统消耗能量的速率，单位是瓦特（W）。若令 ΔE 是在 Δt 时间内所消耗的能量，那么这段时间内的平均功耗 P_{avg} 由下式给出：

$$P_{\text{avg}} = \frac{\Delta E}{\Delta t}$$

瞬时功耗是指 Δt 趋近于 0 时的平均功耗：

$$P = \lim_{\Delta t \rightarrow 0} \frac{\Delta E}{\Delta t} = \frac{dE}{dt}$$

区分能耗和功耗这两个概念对于嵌入式系统的节能技术有着重要作用。一般意义上讲，降低功耗能带来能耗的降低，但单纯降低系统功耗并不总是带来系统能耗的减少。例如，对于一个计算机系统，可以通过降低一半的时钟频率来减少功耗，但是如果这个系统处理事务的时间相应地提高了一倍，那么系统总的能耗可能是相同的。减少能耗和减少功耗是节能研究的两个重要方面，选择其中哪一方面是由具体的上下文决定的。例如，在无线移动应用中，减少能耗能够延长电池寿命，所以减少能耗更加重要；对于其他的一些系统，能量供应充足，这时控制系统温度可能是更为重要的问题，我们需要减少系统的最大瞬时功耗（即峰值功耗）。

功耗优化与能耗优化有着不同的度量标准，因此有不同的优化策略。对面向硬件电路而言，降低芯片功耗能降低嵌入式系统的能耗，因此本章首先分析芯片级的低功耗技术，从 CMOS 电路功耗的组成出发，接着从电路设计到制造过程中多个层次介绍常用的和最新的 SoC 芯片级低功耗技术。另一方面，从嵌入式系统的总能耗角度出发，给出了系统级电源管理和动态功耗管理技术，此处的系统级低功耗技术实际上面向的是降低系统能耗的目标。

8.1.2 SoC 芯片级功耗优化

SoC 芯片是移动智能终端的核心，但随着芯片集成度和设计复杂度的提升，SoC 芯片面临着日益严重的热管理的挑战，功耗指标已经成为芯片设计中的又一关键因素。本小节首先分析 CMOS 电路功耗的组成，然后介绍 SoC 芯片的低功耗技术，涉及从电路设计到制造过程中的多个层次，包括工艺级、版图级、电路级、门级、寄存器传输级、体系结构级和系统级。

1. CMOS 电路功耗的组成

电路的总功耗由动态功耗和静态功耗两部分组成,以 CMOS 反相器为例分析动态功耗和静态功耗,如图 8-1 所示。其静态功耗(P_{STATIC})由器件不活动时的漏电流 I_{LEAK} 引起。而动态功耗(P_{DYN})主要由两部分组成,分别是电路内部节点和电容充放电引起的开关功耗(P_{SWITCH} , 又称翻转功耗)和电源与地通路引起的短路电流功耗($P_{\text{SH-C}}$),它们分别由翻转电流 I_{SWITCH} 和短路电流 $I_{\text{SH-C}}$ 引起。

电路总功耗可以简要地描述为:

$$\begin{aligned} P_{\text{TOTAL}} &= P_{\text{DYN}} + P_{\text{STATIC}} \\ &= P_{\text{SWITCH}} + P_{\text{SH-C}} + P_{\text{STATIC}} \end{aligned}$$

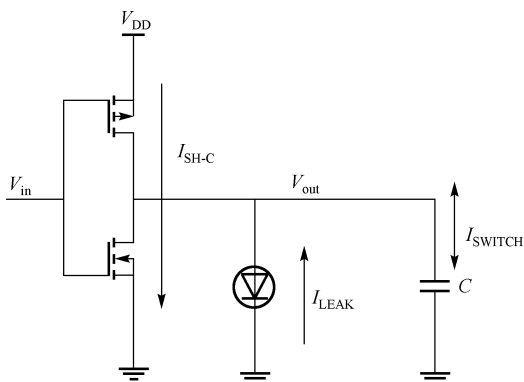


图 8-1 CMOS 反相器功耗示意图

CMOS 器件活动时就会产生动态功耗。当 CMOS 电路的输出端电平发生变化时,会对 CMOS 器件内部电容和节点电容进行充放电,产生开关功耗,其大小正比于总的容性负载。假设输出/输入信号的翻转频率为 f , α 代表信号发生翻转的概率,则开关功耗如下:

$$P = \alpha C_L V_{\text{DD}}^2 f$$

在输入信号变化时,除了产生负载充放电的电流外,还会产生短路电流。当输入电压达到某一值时,在短时间内 PMOS 和 NMOS 会同时开启,从而产生了短路功耗,计算公式如下(其中 I_{SC} 表示反相器的 PMOS 和 NMOS 同时导通时短路电流的大小):

$$P_{\text{SH-C}} = I_{\text{SC}} \cdot V_{\text{DD}} \cdot f$$

静态功耗是电路处于等待或非激活状态时泄漏电流所产生的功耗。在早期的 CMOS 电路中漏电流是可忽略的,但是随着芯片工艺尺寸的减小和阈值电压的降低,内部功耗正在显著提高,在 45nm (及 45nm 以下)工艺时,静态功耗占到整个芯片功耗的 20%~50%。静态功耗是限制手持或电池供电设备寿命的重要因素。静态功耗主要由 4 种漏电流引起:亚阈值漏电流、栅极漏电流、栅至漏极电流、PN 结反偏电流。亚阈值电流是当栅极输入电压小于阈值电压时由于亚阈值传导所产生的静态电流,此时器件工作在弱反型区,有电流从漏极流向源极。在低电源电压应用下,栅电压趋近器件的阈值电压,使亚阈值电流增大。一旦晶体管上电,这些漏电流就会存在,它与时钟频率或开关频率无关。降低时钟信号频率或关闭时钟频率都无法使它减小。但是,通过降低电源电压,或者完全切断晶体管的供电,可以减小甚至消除漏电流。

从 CMOS 电路的功耗来源的分析可以看出,低功耗技术涉及很多因素,如跳变因子、负载电容、电源电压、工作频率、阈值电压及器件尺寸等。低功耗设计就是从这些基本因素出发,在设计各个阶段综合运用不同的策略以消除或降低诸因素对功耗的影响,以取得更好的低功耗效果。

2. 工艺级低功耗技术

采用更先进的工艺能获得更小的晶体管尺寸,有助于减小互连线长度和开关电容;同时,工作电压也会随着工艺进步而降低,从而有效降低电路功耗。同样,多层金属布线可以避免使用大范围连线,减少开关电容以降低功耗。但是多层金属会导致耦合寄生电容的增加,抵消部分降低的功耗。

电源电压随着工艺水平的提高不断降低，为满足性能的要求，阈值电压也随之不断降低。然而，阈值电压的减小会导致泄漏电流呈指数级增长，而且越来越薄的栅氧化层也使得栅沟道泄漏电流不断加大。在 45nm 和更先进的工艺下，泄漏功耗的处理甚至成为芯片设计中的主要部分。针对这一问题，可以采用一些特殊工艺，如绝缘体上硅（SOI, Silicon on Insulator）工艺、多阈值工艺和变阈值工艺等。多阈值工艺在关键路径上采用阈值较低的器件，而在非关键路径上采用高阈值器件，虽然会因此增大延迟，但可换得漏电流功耗的降低；变阈值工艺通过动态地改变衬底偏置电压以改变阈值，同样可降低漏电流功耗。

除此之外，新型器件和工艺也在降低功耗方面有所成就。例如，TSMC 最新工艺采用的 FinFET 晶体管——鳍式场效应晶体管（FinFET, Fin Field-Effect Transistor），是由胡正明（Chenming Hu）发明的新 CMOS 晶体管，现已在 TSMC、Intel、三星等多家代工厂实现量产。在传统晶体管结构中由栅极控制电流通过，但只能在栅的一侧控制电路的接通与断开，属于平面的架构。而在 FinFET 的架构中，栅被制造成类似鱼鳍的叉状 3D 架构，可于电路的两侧控制电路的接通与断开。这种设计可以大幅改善电路控制并减少漏电流（leakage），也可以大幅缩短晶体管的栅长，使其继续向 14nm 及以下发展。

3. 版图级低功耗技术

以前，布局布线技术大多只需要考虑面积和延时的因素，进入深亚微米工艺后，互连线的功耗逐渐成为整个电路功耗的主要部分，在工艺尺寸降到 10nm 下尤其严重，其连线延时甚至会抵消工艺尺寸缩小带来的速度优势，布局布线也就成为低功耗设计需要重点考虑的一个方面。布线时应考虑将开关频繁的路径设为高优先级，同时减小互连线的长度以降低整体功耗。

时钟树（也就是用于传输整个芯片同步时钟的网络）是 SoC 芯片中最大的负载网络，其功耗可达整个 SoC 芯片功耗的 40%。时钟树生成时，可以在保证时序约束的条件下，对时钟树的结构、驱动方式进行选择，并通过缓冲器的插入和尺寸优化来减小功耗。另外，在对同步时钟容差进行分析的基础上，不再追求时钟偏移最小化，而是在保证电路时序的条件下减小功耗。

4. 门级低功耗技术

目前采用的门级低功耗优化方法主要有门尺寸优化和门级多阈值电压技术（Gate-level Multi Vth implementation）。其中，门尺寸优化的基本思想是通过减小器件的尺寸来获得低功耗，但这样做通常会影响电路的性能。作为改进，可以缩小非关键路径的门的尺寸以减小面积和功耗。因此门尺寸优化问题可以转化为满足给定延迟约束条件下的功耗极小化问题。此外，对单元库的结构进行优化也属于门级低功耗方法。

门级多阈值电压技术主要用来降低漏电流功耗。随着芯片集成度的提高，电源电压不断降低，多阈值电压逻辑电路在低功耗设计中发挥着越来越重要的作用。它一方面降低了内部工作电压的逻辑摆幅，使功耗降低；另一方面有效地控制了漏电流的增加，克服了以往因工作电压减少、阈值电压降低而导致的漏电流的增加。

5. 寄存器传输（RTL）级低功耗技术

RTL 级低功耗技术主要通过减少寄存器不希望的跳变（glitch-spurious switch）来降低功耗。这种跳变虽然对电路的逻辑功能没有负面影响，但会导致跳变因子 α 的增加，从而导致功耗的增加。减少毛刺的方法主要是消除其产生的条件，如采用时钟信号同步、结构重构以及时钟门控（Clock Gating）等。

时钟门控技术可以说是当前最有效的低功耗技术，可以减少 30%~40% 的功耗。数字电路中，时钟的翻转必然会引起各时序单元的动作，使得相同的输入值在每个时钟周期都被重复加载进后面的寄存器中，使后面的寄存器、时钟网络和多路选择器产生不必要的功耗。插入门控电路可以将寄存器的

时钟关闭,防止时钟触发寄存器,大幅度降低功耗。时钟门控技术可以作用于局部电路或一个模块,也可以作用于整个电路,作用范围越大,功耗降低越显著,具体方法见 8.2.1 节。

6. 体系结构级低功耗技术

典型的低功耗结构有两种:并行结构和流水线结构,如图 8-2 所示。这两种结构常见于高速电路中用来提高电路吞吐量。此外,在保持电路原有的吞吐量不变时,还可以用来作为降低功耗的手段。

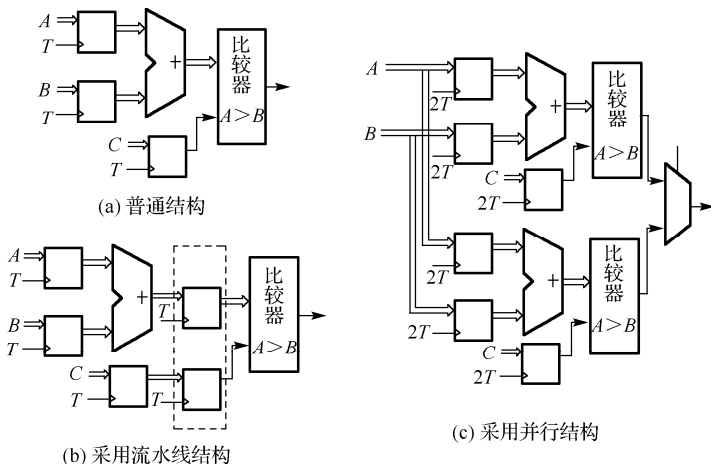


图 8-2 并行结构和流水线结构

并行结构就是把数据流中的一个功能模块“复制”为 N ($N \geq 2$) 个模块,由于有多个模块同时工作,从而提高了吞吐能力。若保持吞吐量不变,则可将工作频率降为原来的 $1/N$,由于一个门的延时和工作电压大致呈线性反比(在超阈值电压下成立),从而工作电压也可以降为原来的 $1/N$ 而性能不变,则理论上功耗大约可降低为原来的 $1/N^2$ 。可见,并行结构可以在保持原有性能的基础上显著降低电路功耗。但是,并行结构增大了电路的面积、电容和延时等,设计时需要权衡各个因素的影响。

流水线结构本质上也是一种并行,然而不同于上面的并行结构,它把指令划分为多个步骤,充分利用每个时钟周期,并行处理多条指令。若工作频率不变,对某个模块的速度要求仅为原来的 $1/N$,则工作电压可以降低为原来的 $1/N$,电容的变化不大(寄存器面积占的比例很小),功耗可降低为原来的 $1/N^2$ 。但是流水线结构设计的复杂性高,在设计中插入的寄存器不但增大了面积,而且增加了时钟负载电容,导致额外的功耗浪费。

7. 算法级低功耗技术

这一级的设计方法主要是对硬件资源的合理利用,以及针对所要实现的功能优化数据信号的编码风格。在进行算法设计时,可以通过因式分解、提公因式等数学方法,找出复用率较高的子函数,将其单独实现成子电路供其他模块调用,以节约硬件资源、减少电路的物理电容。另外,降低开关活动因子是降低功耗的一个有效方法,尤其对结点电容大的信号线更是如此,比如总线。现在的大型芯片中总线的数据线和地址线一般都比较长,每条线都需要驱动大负载,通常占总功耗的 15%~20%,有的甚至达 70%以上。我们可以采用合适的编码方式来降低开关活动频率,如格雷码。格雷码是通过对二进制数编码,实现连续的两个二进制数之间只有一位不同,这样传输连续变化的数据时在总线上只有一位发生变化,总线的翻转活动大大减小,从而降低功耗。其他常用的编码还有独热码和二进制补码等。

8. 芯片系统级低功耗技术

降低功耗在芯片设计流程中进行得越早越好，这样可以有效地降低功耗预算，避免重新设计带来的成本浪费。芯片系统级（注意：我们这里所说的系统级是指站在全芯片的角度来看低功耗设计方法，请读者不要把这里的系统级和 8.3 节的系统级低功耗设计相混淆，后者是指站在整机产品设计方案的角度来优化功耗）低功耗技术主要有：多电压域、电源门控、动态电压频率调节、自适应电压调节技术等，这些技术虽然在方法上归为系统级，其具体实现必须依赖相应的电路层面的支撑，我们将在 8.2 节重点介绍其电路实现技术。

8.1.3 嵌入式系统级功耗优化

一般而言，一个嵌入式系统的主要能耗部件包括嵌入式微处理器（SoC 芯片）、内存、LCD 液晶显示器及背光、Wifi 模块、电源转换部件以及 USB、SD 卡等外围设备电路。以一个典型的平板电脑嵌入式应用系统为例，其能耗部件的耗电情况如图 8-3 所示。其中，LCD 及背光占到了 35%~48%，处理器产生的功耗占到了整个系统的 23%~30%，内存占 14%，系统的其余部分占用剩余的 15%~20%。

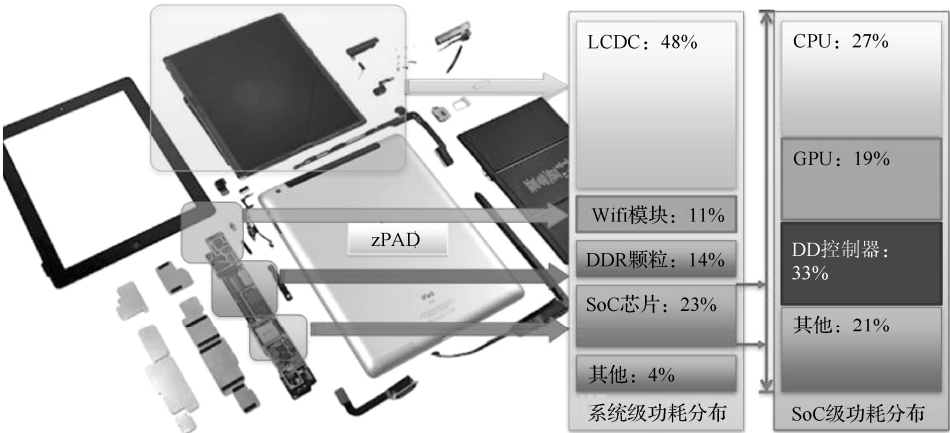


图 8-3 嵌入式系统能耗部件的耗电统计

在这些能耗部件中，一些部件取决于芯片设计和电路本身，而有些部件拥有多种工作状态——多种可控的能耗状态。前者可以通过芯片级低功耗技术（对嵌入式系统设计人员来说简化为芯片选型）和硬件电路设计来寻求功耗的降低，后者的实现借助操作系统辅以相应的机制设计和策略应用能够得以实现。

总之，欲完成一个低功耗性能突出的嵌入式系统的设计，首先要在尽可能考虑软硬件协同设计的基础上设计出低功耗硬件平台，其次要依靠优秀的软件低功耗技术最大程度地发挥硬件平台的低功耗特性。

嵌入式系统级功耗优化的定义：

- 从系统软硬件的整体出发寻求降低系统、子系统或设备功耗的设计方法，而并非着眼于通过内部机理的改进降低单个设备或子系统功耗。
- 可以在硬件系统、操作系统、应用程序，甚至用户操作等多个系统层面上实施。
- 通过与系统运行过程以及各类请求负载的相互作用来实现降低功耗的目标。

相对于电路级、结构级等低功耗技术而言，嵌入式系统级功耗优化技术位于整个电子系统的较高层次，因此可以充分利用应用的信息；而随着操作系统的迅速发展与广泛应用，系统级功耗优化也获

得了比以往更多的观察与实现手段。丰富的信息量与技术手段使得系统级功耗优化可以十分有效地根据应用情况取得功耗与性能的良好平衡，因此系统级功耗优化技术在最近几年获得了较多的关注。随着微处理器性能以及内存容量的不断提高，各类嵌入式系统的计算能力增长迅速，嵌入式系统级功耗优化技术在内部也有向操作系统、应用程序等软件级别转移的趋势。

嵌入式系统级功耗优化技术的内容十分丰富，所涉及的知识背景也较为庞大。在系统级，能量消耗主要来源于 4 部分：①处理单元；②存储系统；③显示系统；④内部连接与通信单元。嵌入式系统级功耗优化技术在保证各部分性能要求以及交互效应达到平衡的同时，使得这 4 部分的能耗最小化。

表 8-1 是常用的低功耗技术及其简要的工作原理，其中上半部分是 SoC 芯片级低功耗设计方法，下半部分是嵌入式系统级低功耗设计方法。

表 8-1 常用的低功耗技术工作原理

SoC 芯片级低功耗设计方法	工作原理
时钟门控 (Clock Gating)	控制时钟的开启和关闭，在模块不工作时关闭时钟，在需要工作时打开时钟，从而降低触发器总的翻转率，以降低芯片的动态功耗
电源门控 (Power Gating)	根据应用需要有选择性地关断芯片中暂时不需要的一个或多个电压域，减小芯片的动态功耗和静态功耗
多电压域 (Multi-Voltage)	按照芯片功能和应用需要，将不同的逻辑模块放置在不同的电压域中，这些电压域由电源管理模块分别独立供电，使得不同的逻辑模块在不同的电压下工作
自适应电压调节 (Adaptive Voltage Scaling)	通过建立一套片上闭环反馈系统，根据芯片工作时实时的环境影响和工艺偏差影响自适应调节电压以降低功耗
多阈值技术	低阈值电压单元漏电流较大，但工作频率高、速度快；高阈值电压单元漏电流较小，但工作频率低、速度慢。在时序关键路径上使用标准阈值电压和低阈值电压单元来满足时序要求，在非时序关键路径上使用标准阈值电压和高阈值电压单元来满足漏电功耗最小化
衬底偏压调节 (Body Bias)	晶体管的阈值电压与其衬底偏压有关，而亚阈值电流又与晶体管的阈值电压有关，可以通过器件的衬底偏压对亚阈值电流进行控制。自适应衬底偏置电压技术则在芯片内部设计反馈机制，自动将目标电路的衬底偏置电压调整至当前条件下的最佳值，从而完成待机模式下漏电功耗的最小化，并对工艺和温度变化对电路造成的影响进行补偿
低功耗工艺库设计	对于工艺库中常用的逻辑单元进行优化，采用新的低功耗电路结构，在满足性能的要求下尽可能降低功耗
嵌入式系统级低功耗设计方法	工作原理
DPM (动态功耗管理)	动态电源管理系统，有选择地关闭空闲的系统部件或降低其性能，在需要服务时再重新打开，以减少功耗
DVS (动态电压调节)	在任务运行过程中根据工作负载的应用需求（即任务完成时间）来动态调节设备（以处理器为主）的工作电压
DVFS (动态电压频率调节)	采用一定的监测手段和预测算法，检测当前应用的性能需求，动态地调节电源电压和工作频率，达到降低功耗的目的

8.2 SoC 芯片级低功耗设计方法

动态功耗是 SoC 芯片功耗的主要组成部分，但随着深亚微米工艺的发展，之前微不足道的漏电流功耗呈指数级增长，甚至有超越动态功耗的趋势，这也使得新兴低功耗技术的研究显得更加重要和紧迫。本节主要介绍 SoC 芯片层面几类重要的和新兴的低功耗技术。

时钟门控技术是一项传统的降低动态功耗的技术，可有效降低芯片内部频繁的信号翻转造成内部功耗和开关功耗的增加，也会降低时钟树上的功耗消耗。

此外，减少电源电压是降低动态功耗最有效的方法，因为动态功耗和电源电压的平方成正比。不过，降低电源电压会使得延迟增加、性能下降。作为折中，可以在阈值电压不变的情况下，采取多电源电压（MSMV，Multi Supply Multi Voltage）的方法。即在系统的关键时序路径上采用较高的电源电压保证整个系统的性能，而在其他路径上采用低的电源电压以减少功耗。不过，需要在不同电压域之间加入电平转换电路。同时结合电源门控技术则能极大降低模块不工作时的静态电流。结合动态电压频率调节技术，则能根据电路工作负载的大小调节供电电压和频率，进一步降低动态功耗。若采用自适应电压频率调节技术，则直接由芯片内部的监测单元和调节系统完成电压频率的自动调节，无需嵌入式系统工程师对芯片的工作过程进行调度，降低了使用难度和系统级代价。

8.2.1 时钟门控

时钟门控技术作为一项传统的降低动态功耗的技术被广泛应用于现代数字集成电路设计中，它用一个控制信号控制时钟的开启和关闭，从而通过降低触发器总的翻转率达到降低功耗的目的，其特点为实现简单，并且十分有效。时钟门控技术可以作用于局部电路或一个模块，也可以作用于整个电路。在局部电路应用中现在已为标准流程，基本上所有的商业化 EDA 工具，如新思公司（Synopsys Inc）的综合工具 Design Compilier，Sequence Design 公司的 Power Theater 工具，以及 Cadence 公司的 SoC Encounter 工具都支持自动插入门控时钟单元的功能，同时调整时钟树网络，以解决门控时钟单元带来的时钟偏移（Skew）和延时（Delay）。在模块级应用中，需要在 RTL 中手动设计来控制整个模块的时钟开启和关闭，在模块不工作时关闭时钟，在需要工作时打开时钟。

图 8-4 (a)所示结构为传统的带使能端的触发器设计，通过使能信号 EN 来控制当时钟信号来临时，寄存器采样新值 D 还是保持原来的值 Q；图 8-4(b)所示结构为应用时钟门控技术的设计，通过用 EN 信号控制时钟信号的开关，在 EN 信号无效时，寄存器的时钟端将保持一个定值，D 端的数值将不能传到 Q 端。

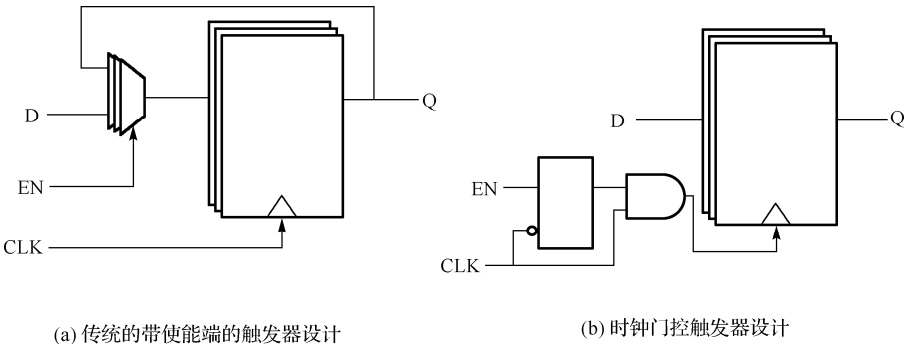


图 8-4 传统的带使能端的触发器设计和时钟门控触发器设计

在逻辑综合过程中对 RTL 代码中插入门控时钟的方法是：判定一组寄存器是否共用一个使能信号，此信号用来决定当有效时钟来临时寄存器是否能读入新的值。传统的方法是用这个共用的信号来控制连接在寄存器 D 端的选择器，或者连接到具有时钟使能端的寄存器的时钟使能端。运用门控时钟技术，综合工具会找到这些共用的控制信号，用它们控制时钟门控单元。因此，如果一组寄存器共用一个使能信号控制门控时钟单元，当此使能信号无效时，这组寄存器几乎不消耗动态功耗，且时钟门控单元本身消耗的功耗很少。下面以一段简单的 RTL 代码来说明此过程，代码描述了一个 3 位的计数器。

```

module counter (CLK,RST_N,INC,COUNT)
input CLK;
input RST_N;
input INC;
output [2:0] COUNT;
reg [2:0] COUNT;
always@ (posedge CLK or negedge RST_N)
begin
  if (~RST_N)
    COUNT <= #1 3b0 ;
  else if (INC)
    COUNT <= #1 COUNT + 1 ;
end
endmodule

```

计数器有异步复位信号 RST_N，当 RST_N 拉低时，计数器复位（归零）；RST_N 置高时正常计数，此时当 INC 信号为高时，计数值在每个时钟周期加 1，如果 INC 为低，计数值保持不变。采用传统的选择器综合方法，综合结果如图 8-5 所示。此时时钟信号直接连接到每个寄存器的时钟端，这就意味着在 INC 信号为低，即寄存器的输出值通过选择器返回到寄存器的 D 端（数据输入端）时，时钟端的信号仍然在不停地跳变。

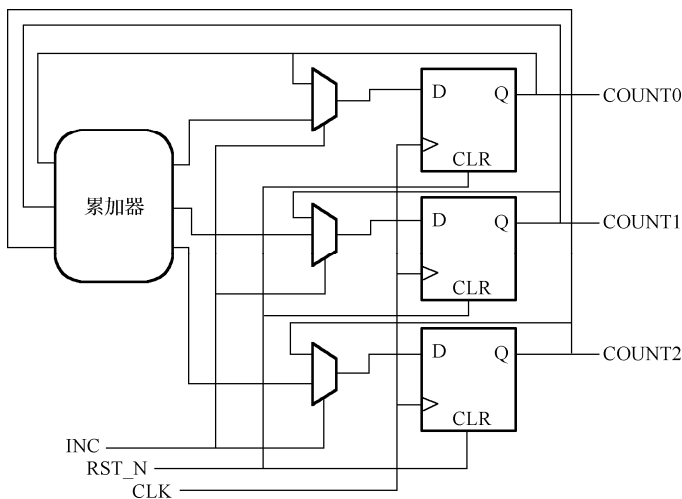


图 8-5 传统方式下实现的三位计数器

图 8-6 是用门控时钟技术实现的相同电路（三位计数器）。两个电路很相似，只是后者在时钟网络上加入了时钟门控单元，只有当 INC 信号为高时，时钟信号才能穿过时钟门控单元到达寄存器的时钟端。当 INC 信号为低时，寄存器没有时钟翻转，所以将如传统设计一样保持原来的值。这样就可以去掉传统设计中的寄存器前级的 3 个选择器，如果在多个寄存器的实现中则会显著减小面积。

常见的时钟门控单元分为两种：锁存器类型（Latch-based）和非锁存器类型（Latch-free）。非锁存器类型只需要一个简单的与门（AND Gate）或者或门（OR Gate），如图 8-7 所示，具体使用与门还是或门取决于寄存器是由上升沿触发还是由下降沿触发。应用此结构的时钟门控单元时，要注意使能信号要在时钟信号的非触发沿变化，防止时钟信号的变化在切换时被截断，或者产生毛刺，如图 8-8 所示。

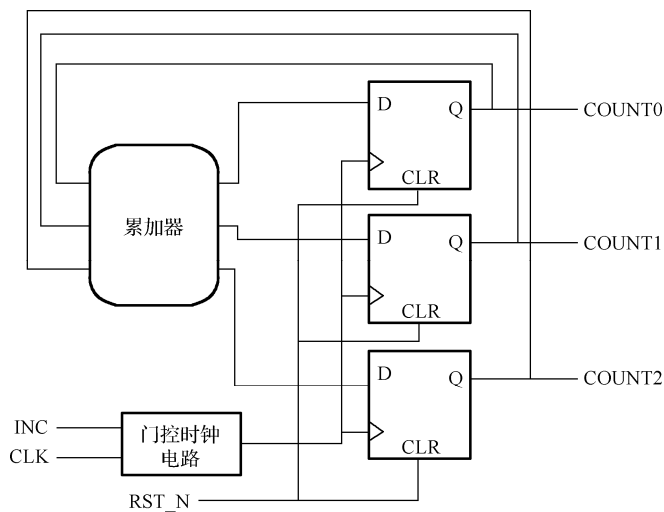


图 8-6 使用门控时钟技术后的三位计数器

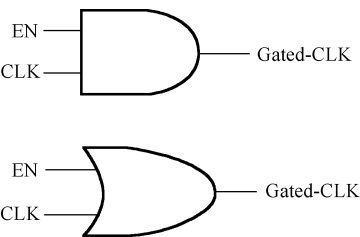


图 8-7 非锁存器类型的时钟门控单元

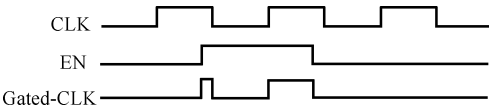


图 8-8 非锁存器类型的时钟门控单元会产生毛刺

非锁存器类型的时钟门控单元对于使能信号的时序有一定的要求，对于单时钟寄存器设计不太适合。锁存器类型的时钟门控单元加入了电平敏感的锁存器，如图 8-9 所示，用来将使能信号从时钟的触发沿保持到非触发沿，对于使能信号的时序没有特殊的要求，如图 8-10 所示。

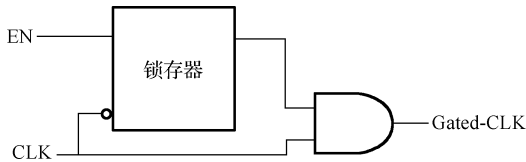


图 8-9 锁存器类型的时钟门控单元

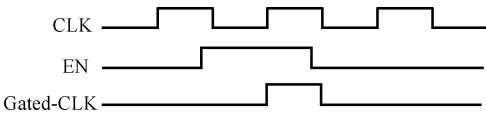


图 8-10 锁存器类型的时钟门控单元参考时序

8.2.2 多电压域技术

多电压域（Multi-Voltage）技术是按照芯片功能和应用需要，将不同的逻辑模块放置在不同的电压域中，这些电压域由电源管理模块分别独立供电，使得不同的逻辑模块可以在不同的电压下工作。例如，某些性能要求不高的模块可位于低电压域中，而性能要求较高模块的供电电压相应较高。多电压域技术也是动态电压频率缩放、静态电压缩放、自适应电压缩放设计的基础。

电路采用多电压域技术会给设计带来一些新的挑战：

- 信号在不同电压域之间传递，需要插入电平转换器（Level Shifter）实现电平转换。
- 由于芯片各个模块会工作在多种电压下，因此在各种电压下的时序要求都要满足，因此加大了静态时序分析（STA，Static Timing Analysis）及时序签核（sign off）的复杂度。

- 电源网格 (power grid) 的布局规划、模块接口单元的电源布线等都变得更复杂。
- 板级上需要更多的电压调节器来提供各种不同电压, 增加了板级设计的复杂度。
- 由于各个模块电压不同, 模块间上电/下电顺序也需要仔细设计, 避免电路出现死锁。

当一个信号从低电平电压域进入高电平电压域时, 可能出现由于 PMOS 晶体管栅极所加电压过低而导致该逻辑门无法完全关断的情况, 电平转换器的使用可防止这种情况下出现的不正常的漏电电流; 其次, 因为信号必须在不同电压域中进行翻转, 电平转换器能保证这些信号线的翻转时间与延时计算正确, 从而得到正确的时序信息。电平转换器实现不同电平之间的转换, 属于模拟电路, 而且由于模拟电路设计问题, 这些电平转换器都是单向的, 从高电平到低电平转换或从低电平到高电平转换。

当一个信号从高电平电压域进入低电平电压域时, 可不用高电平向低电平转换电路, 也可以采用一个反相器或缓冲器实现, 典型的高到低的电平转换器如图 8-11(a)所示。电平转换器放置在低电压域中, 其栅极上可以有一定的输入过压, 输出转换为低电平。

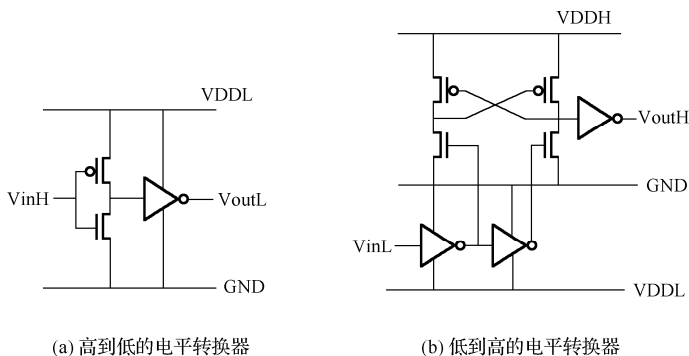


图 8-11 电平转换器

低到高的电平转换器一般需要专门的转换单元, 因为低电平电路的输入信号不足以驱动高电平电路中的 NMOS 管, 从而将导致电路的上升时间和下降时间变得很长, 电路速度变慢。一个简单的低到高的电平转换器如图 8-11 (b)所示, 通过输入和输入的反相信号驱动一个放大器。若低电平低至亚阈值电压, 则需要特别设计亚阈值电平转换器。

8.2.3 电源门控技术

随着工艺技术的发展, 由漏电流所产生的功耗所占的总功耗比例越来越大, 休眠模式下漏电流功耗成为设计者必须考虑的因素。如果希望在休眠模式下尽量节省功耗, 最好的办法是将处于休眠模式状态的模块的供电电源关断, 同时保持其他模块正常供电, 这种技术叫电源门控 (Power-Gating) 技术。与仅降低动态功耗时钟门控技术相比, 电源门控更进一步地消除了静态功耗。此外, 电源门控技术影响各模块之间的相互连接, 安全进入和退出电源门控模式会增加很多额外的操作, 其状态切换需要严格设计; 而时钟门控技术不影响电路的功能, 除了模块级时钟门控需要略微修改 RTL 外, 基本的时钟门控可使用 EDA 工具自动实现, 它在设计和实现上可以是对设计者透明的。

电源门控一般有两种方法来实现:

- 外部电源门控 (external power gating)。举个例子, 一个 SoC 系统在板级上有 CPU 的专用电源, 这个电源只提供电压给 CPU。外部电源门控技术即为 CPU 在非活动状态时关闭这个电源以使漏电流功耗减小到零。但这种方法在重新上电时也需要最长的时间, 以完成对重新供电和数据的重新加载。
- 内部电源门控 (on-chip power gating)。内部电源门控是指在芯片内部用一些专门的逻辑单元 (如电源门控单元) 来控制所选模块的供电情况。

外部电源门控技术与内部电源门控技术均能实现将电压域中电压关断从而最大限度地减小漏电功耗的目的，但在物理实现过程中，内部电源门控技术要复杂得多。

内部电源门控技术有两种实现方法，它们分别使用不同的电源门控单元：粗粒度和细粒度电源门控单元。

细粒度电源门控单元是在工艺库中每个标准单元结构的电源/地和构成逻辑的晶体管之间插入门控晶体管，以切断电路的漏电电流。因为插入的晶体管要能提供所有情况下该单元所需的电流，而且为了防止对设计性能的影响，其宽长通常设计得很大。因此，使用细粒度门控单元的电源门控设计，不但大大地增加芯片面积、紧缩布线资源，还在一定程度上加大延时，影响电路的性能；如果其宽长比太小，则会影响系统的抗噪声性能，降低系统可靠性，甚至会导致电路无法正常工作。当然，细粒度电源门控单元也具有优点：每一个单元可以有很好的模拟性能，包括对直流压降（IR Drop）的影响和时序的影响，因为它们都集成在同一个标准单元中，可以用传统的方法实现电源门控物理设计。图 8-12 中左图是细粒度电源门控单元的结构示意图。

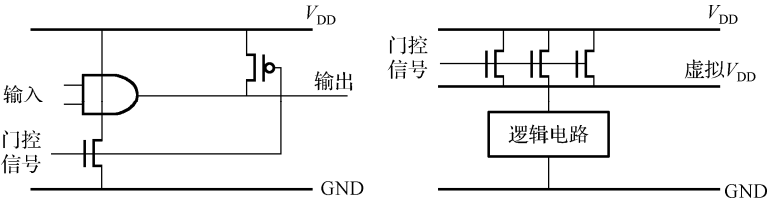


图 8-12 细、粗粒度电源门控结构图

粗粒度电源门控单元是利用门控单元控制整行甚至多行标准单元电路与电源/地线之间的连接，从而减小每个单元的面积和多余的单元端口。门控单元的晶体管尺寸的选择比较关键，通常其宽长比较大，它的结构设计比细粒度电源门控单元更复杂，但显而易见，使用粗粒度电源门控单元比使用细粒度电源门控单元的设计面积明显小很多。图 8-12 中右图是粗粒度电源门控单元的结构示意图。

电源门控设计中不论使用外部电源门控还是内部电源门控，均会遇到一个问题：被电源门控的模块在门控过程中，因为它们的输出信号变化缓慢，其信号值有很长一段时间处于阈值电压附近，会造成相邻工作系统上 N 管、P 管常开，造成大量的直通电流（crowbar current）。为了解决这个问题，在模块相邻的接口之间需要添加隔离单元（isolation cell，ISO）。当模块电源关断发生时，使能该模块的隔离单元，使其他模块不会受到输入的中间电平影响。隔离单元设计思想是把这些不定的输出信号钳位到一个特定的合法值。隔离单元有 3 种类型：钳位到“1”、钳位到“0”和锁存到最近值。前两种隔离单元的原理图如图 8-13 所示。

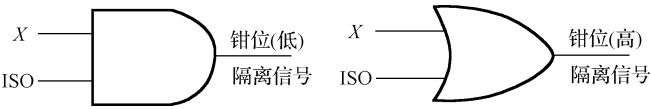


图 8-13 隔离单元示意图

隔离单元的缺点是会增加电路延时，对某些关键路径而言，增加延时会降低设计的性能。另一种不会增加很大延时的隔离技术是使用上拉或下拉晶体管，但此法会引入端口上的多驱动问题，需仔细规划模块掉电和隔离使能的次序以防止竞争的发生。虽然使拉高或拉低晶体管是相对的弱驱动逻辑，但也会引起总线竞争、产生过大的电流而导致错误。如图 8-14 所示是上拉、下拉晶体管的结构示意图，其中左图表示上拉晶体管，当“ISOL”信号为高电平时，电源关断模块的输出信号被钳位到“0”，为低电平时，输出信号正常；相反，右图所示为上拉晶体管结构示意图，当“ISOLN”信号为低电平时，

电源关断模块的输出信号被钳位到“1”，为高电平时，输出信号正常。此外，这种多驱动的隔离方式也会给测试带来困难。图 8-14 仅仅是隔离单元的示意图，实际使用中，需要通过一系列门控晶体管（通常为菊花链形式）进行电源的隔离，以防止上电过程中出现过冲现象。

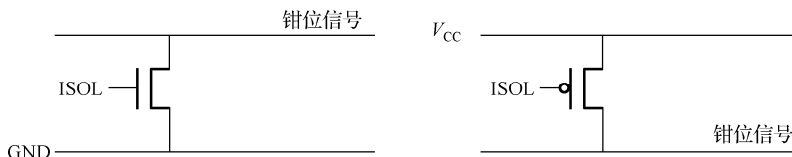


图 8-14 上、下拉晶体管隔离单元示意图

8.2.4* 动态电压频率调节和自适应调节

1. 动态电压频率调节（Dynamic Voltage Frequency Scaling）

传统的 SoC 芯片工作在固定的电压和频率下，无论工作负荷繁忙与否，都使得电路一直保持全速运行，因而浪费了很多能量。动态电压频率调节技术则根据工作负载的大小需求调整供电电压和频率，大大降低了动态功耗。动态电压频率调节技术需要得到硬件设备的支持，一般会与其他技术（比如多电压域、电压关断技术等）结合起来，同时配合系统级的软件调度，可以发挥其降低功耗的效果。

动态电压频率调节方法根据采集系统负载方式的不同可以分为两种——基于软件调度和硬件监测。软件方法通常被归类为嵌入式系统级的低功耗技术，将在 8.3 节中介绍。无论是基于软件调度还是硬件监测，都会采集与系统负载相关的信息。软件实现是根据核心函数调用的频度不同使用不同的算法来判断系统的负载。硬件实现则通过采集一些核心信号中断线、Cache、内存总线的使用情况等，计算当前的系统负载。这样，一方面确保了负载计算的准确性；另一方面减轻了 CPU 用于负载跟踪与性能预测的负担。

动态电压和频率调节方法遵循相应的工作流程，具体如下：

- 采集与系统负载有关的信息，计算当前的系统负载。该过程可由软件实现，也可由硬件实现。
- 根据系统的当前负载，预测系统在下一时间段需要的性能。根据具体应用来选择不同算法。
- 将预测的性能转换成需要的频率，从而调整芯片的时钟设置。
- 根据新的频率计算相应的电压，并通知电源管理模块调整供给 CPU 的电压。这需要特别的电源管理芯片，比如 Freescale 公司的 MC13783 或者 NS 公司的支持 PowerWise 特性的系列电源管理芯片。它们能够支持微小的电压调整（25mV），并且能在极短的时间内（几微秒到几十微秒）完成电压的调整。

基于硬件监测的 DVS 方法主要分为两种——基于开环控制的 DVS 和基于闭环调节的 DVS。对于后者，自适应动态电压调节（AVS，Adaptive Voltage Scaling）是其中的典型代表。目前许多芯片公司支持基于硬件监测的动态电压和频率调节，比如 Intel 公司的 SpeedStep 技术、AMD 公司的 NCQ 技术及 ARM 公司的 IEM（Intelligent Energy Manager）技术。

基于开环控制的 DVS 是传统的方法，即利用电压频率查找表来完成电压和频率的调节。首先，电源电压调节模块根据来自处理器内核中时钟管理单元的频率请求，从内部的电压频率查找表中取出相应的电压值，并传送给电压输出控制模块。然后使用定时器来延迟时钟管理单元对于频率的确认，直到电压稳定为止。

2. 自适应电压调节技术（AVS，Adaptive Voltage Scaling）

随着微电子制造工艺的特征尺寸向超深亚微米缩小，工艺偏差（包括阈值电压失配、宽长比失配等）、环境扰动（温度变化、电源电压波动、噪声）等因素对芯片性能的影响越来越大，在电路实际工

作过程中，必须考虑到各种因素对芯片的不利影响，以保证其能在最坏情况下正常工作，“最坏情况”即为对芯片正常工作造成负面影响的各不利因素同时出现的情况。由于在数字集成电路设计中，PVT 变化、噪声等因素对芯片的影响都可归结为关键单元和特殊路径延时特性的变化，最坏情况即电路工作时序余量最小、时序最为紧张的情况。影响芯片时序余量的因素主要包括工艺波动、电压波动、温度波动和老化 4 个方面，通常称为 PVTA（Process, Voltage, Temperature and Aging）波动，其中前 3 个因素（PVT 波动）影响最大，在先进工艺下的电路设计中需考虑其影响。

为了保证数字电路正常工作，必须保证关键路径在单时钟周期内工作正确。电路设计中，关键路径需要通过静态时序分析（STA, Static Timing Analysis）工具找到。但是，由 STA 工具分析出的关键路径只是拓扑学上得到的，电路在实际工作中并不一定有实际数据通过，即使有，也与激励密切相关，即并非每组激励都会引起关键路径的数据变化。这就使某些时刻，电路实际的工作电压并非需要如所推测出的那样高，关键路径的电压因素是影响时序余量的重要因素。

最坏情况综合考虑了电路中可能存在的电压抖动、温度波动、栅长、掺杂波动、时钟抖动、耦合噪声等各种因素的不利影响，但这种种不利因素的组合实际却很少甚至不会发生。然而为了确保芯片能够应对不确定的电路模型和最差情况下的电路变化，设计者往往会留出足够的时序余量，选择让电路工作在一个相对保守的电压下，这就使电路的实际工作电压不能降低到一个更低的水平。因此，需要这样一种 AVS 技术，可以在保证性能和功能正确性的前提下，尽可能地释放为保证芯片内部电路始终正确工作而预留的时序余量，以降低电源电压，节省动态功耗。

AVS 技术的基本思想是将 PVT 因素对电路的影响归为延时特性的变化，在电路中加入监测单元监测延时信息，根据电路延时调节电压，降低设计阶段预留的电压余量，从而使芯片处于最佳能效状态，降低功耗。AVS 技术利用硬件调节电压，无需软件执行，调节效率高且不影响系统性能；它可以降低工艺偏差、温度及老化对芯片的影响，提高芯片的产量与质量，还可以降低芯片电源 IR-drop 的影响；另外它只需在 SoC 中添加少量监测电路即可实现，额外的面积及功耗消耗都不大。

AVS 的监测单元有 3 种常见结构，第一种是环形振荡器，第二种是复制关键路径，第三种是片上时序监测单元。第一种方法较为简单，在芯片中放置工作在相同环境（工艺、电压、温度相同）下的环形振荡器，由于其振荡频率在不同的 PVT 下差别较大，因此可以根据其振荡频率粗略估计出整个芯片的工作环境好坏，进而进行监测。

第二种监测方法是复制关键路径，手动设计一条或数条路径来模拟真实的关键路径，通过监测该路径的时序变化情况来反映真实的 PVT 状况。如图 8-15 所示为一种长度可调的复制关键路径结构^[1]，由反相器链和多路选择器构成，复制的关键路径能针对不同的环境进行不同长度的设定。选取国产高性能处理器 Unicore 作为平台，实现基于复制关键路径的 AVS 系统，在系统频率发生变化时，电压首先调至与目标频率所对应的安全工作电压，然后再根据复制关键路径的监测结果自适应地调节工作电压。在 HSIM 和 VCS 联合仿真平台下得到加入了复制关键路径机制的 Unicore 相比于原始 Unicore 在恒定 1.2V 下具有明显的功耗节省，其中 FF 工艺角、25℃下节省了 42.2%的功耗，TT 工艺角、25℃下节省 38.3%的功耗。也有研究采用反相器、与非门和线延迟等的组合设计可配置的延迟链，用来模拟真实的关键路径。

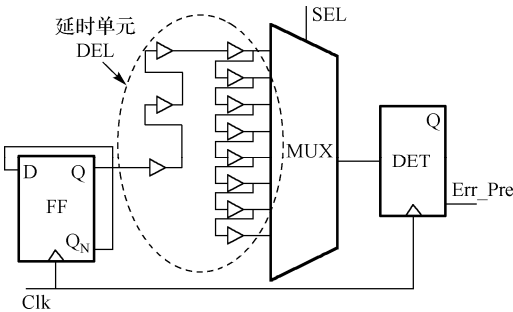


图 8-15 复制关键路径监测示意图

第三种方法效果更好，通过片上时序监测电路来监视可能出错的各个关键路径，将各类扰动的影

合到延迟特性中,并在电路延时超过当前允许值时自适应地调整 VDD 的值来补偿各类扰动的影响,从而在避免电路出错的前提下使芯片工作在最低的临界电压 VDD 下。时序监测单元(In-situ timing monitor)有时又称为错误检测单元(EDC, Error Detection Circuit),典型的代表是 Razor 结构和 Canary Flip-flop,主要分为两类:时序预测型/不出错(Error Prediction / Always correct)和出错改错型(Let fail & correct)。

预测型的典型代表是 Canary Flip-flop,通过替换芯片关键路径上的普通触发器,在时序较为紧张但尚未出错时发出时序预警。若采用监测复制的关键路径(Critical Path Replica)方法,则无需改动原电路结构,配合以电压频率调节即成为一种有效的 PVT 偏差弹性设计方法,可释放组合逻辑路径上的时序冗余,减小动态功耗。例如,AMD 在 ISSCC 2015 上发布了一款 28nm x86 APU^[2],采用接近出错的监测单元来监测复制关键路径,结合 AVFS 能在 0.8~1.3V 电压范围内有效获得功耗收益。这种基于时序预测的不出错在线监测方法仅需要留有很小比例的时序余量,由于不需要额外的改错机制,具有易于实现、电路面积代价小、消耗的额外功耗小的优势。

出错改错型的典型代表是密歇根大学提出的 Razor 和 Razor II 原地错误监测方法^[3,4],将电压降低直至电路时序出错,并利用原地恢复或者上层恢复机制来恢复芯片的正确工作状态。然而, Razor 结构存在短路径问题,需要在延时较短的路径中插入大量的缓冲器以保证 Razor 中上下两个触发器监测的是同一周期的信号,这就使组合逻辑部分面积增大较多。此后,ARM 和 Intel 两大 CPU 公司相应提出基于在线时序监测的偏差弹性设计方法,并应用在处理器中,用流水线重新执行的方法进行错误恢复。其中在密歇根大学和 ARM 公司提出的一种“出错纠正”的方案中,设计了能够监测局部和全局偏差的时序监测单元电路,用来替换 CPU 流水线中的普通触发器,并设计了一套基于流水线刷新的纠错机制和相应的电压/频率调节机制。

该方法目前在常规电压区应用较多,整体结构如图 8-16 所示,在主芯片的一系列关键路径末端插入监测单元,以此来实时监测芯片中时序最为紧张的关键路径,当时序发生违规时,监测单元产生预警/出错信号,输送给芯片中的自适应电压频率调节模块来调节芯片的工作电压和频率。若采用出错型监测单元还需进行错误恢复。

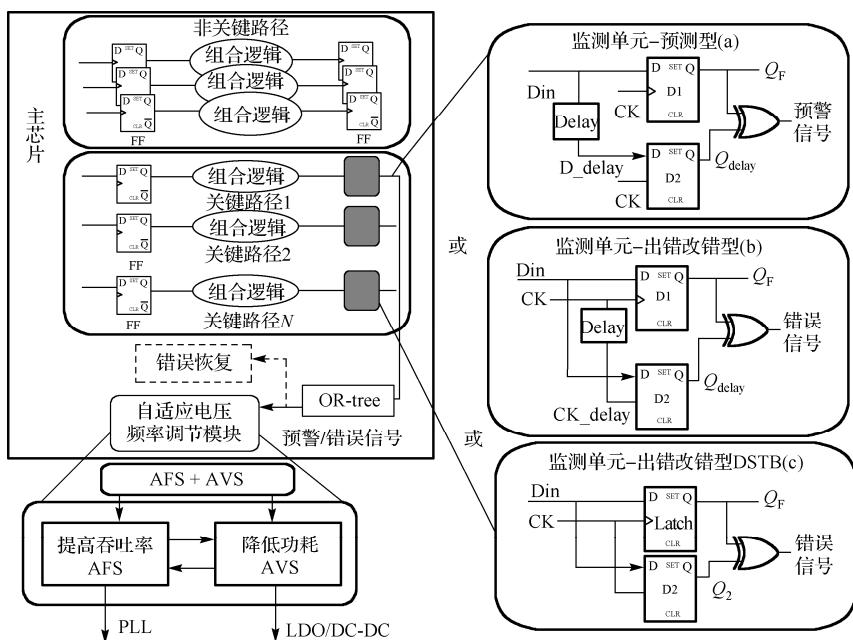


图 8-16 流水线乘法器 AVS 系统动态电压调节效果

典型的常压下所采用的监测单元如图 8-16 右侧(a)和(b)所示，主体由两个并行触发器（或一个触发器和一个影子锁存器）、延时单元和异或门组成。具体可分为错误预测（Error Prediction）和错误检测（Error Detection）两类电路，前者在时序快要出错前发出告警信号，后者是时序真正出错时发出错误信号，二者的构成很类似。错误预测型监测单元结构如图 8-16(a)所示，第二个影子触发器 D2 的输入数据被 delay 单元延迟一段时间，其效果等同于增大了 D2 的建立时间，因此当 VDD 降低或 PVT 波动致使此关键路径时序紧张时，时序违规将首先出现在 D2 上，此时 D2 与 D1 输出不同，异或后产生错误预测信号，只有延时进一步恶化时才会导致正常数据路径上的触发器 D1 出现真正的时序错误。错误检测电路如图 8-16(b)所示，与图 8-16(a)不同的是将影子触发器 D2 的时钟输入延迟一段时间，因而 D2 总是有比 D1 更多的时间采样数据，将 D2 的输出与常规触发器的输出进行比较，若不一致则说明产生了时序错误。

如果监测单元中的主寄存器改为锁存器，即为另一种出错改错型的监测单元，称为双采样监测单元（DSTB），如图 8-16(c)所示，它通过在时钟下降沿采样直接将时钟高电平时间作为监测窗口，从而避免了插入额外的延时单元，但通常需要额外的时钟占空比调节电路来调节其监测窗口 Tw 的宽度。

芯片实际可能有不止一条关键路径，因而应放置多个错误预测与检测电路，其结果综合取“或”后作为最终的错误预测/检测结果送入电压调节单元。不同的应用场合对芯片工作的鲁棒性要求不同，应该使用不同的电压调节策略：在要求极其苛刻的场合，第一次产生错误预测信号即可停止降低 VDD；在一般要求下，可以在第一次有错误检测信号产生时即停止降低 VDD，并将 VDD 略微提高使该信号重新变为 0；更进一步，在要求宽松的应用下，若容许一定的出错比率（例如 0.1%），则可在第一次出现错误检测信号时继续降低 VDD，即以一定的出错比率换取功耗的最小化。

图 8-17 为东南大学设计的基于错误检测和改错的片上监测电路和 AVS 调节系统^[5]，包括了三级流水线乘法器为主电路、错误纠正控制信号产生电路和动态电压调节模块。

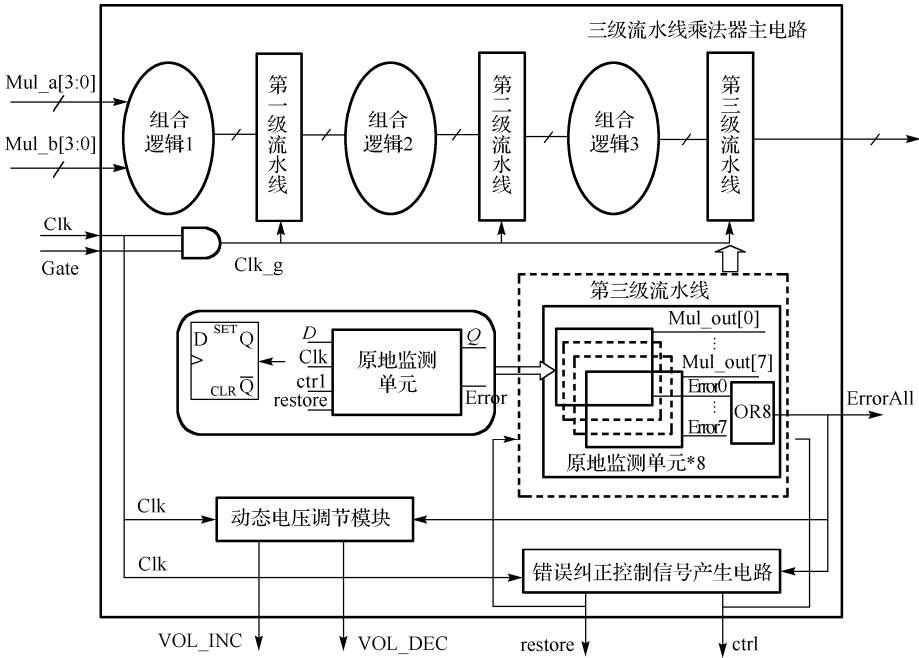


图 8-17 基于片上时序错误检测和改错的 AVS 调节系统

普通的三级流水线乘法器的输入为两个 4 比特乘数——Mul_a[3:0]和 Mul_b[3:0], 输出为 8 比特乘法运算结果 Mul_out[7:0]。乘法器 RTL (Register Transfer Level) 级设计完成后, 用逻辑综合工具 Design Compiler 对其时序进行分析, 筛选可能的关键路径。经分析比较得出, 第二级流水线寄存器输出端到第三级流水线寄存器输入端之间的各条路径时序最为紧张。所以, 用原地监测单元替换第三级流水线上所有的普通 D 触发器来实时监测乘法器工作情况。保证了时序最为紧张的第三级流水线上不出现时序错误, 可以认为整个流水线乘法器的时序是安全的。最后一级流水线上的 8 个普通 D 触发器被原地监测单元所替换, 对 8 个监测单元的监测结果进行或操作, 得出一个总的错误标志信号 ErrorAll 作为告警信号并送到错误纠正控制信号产生电路产生控制信号 restore 和 ctrl。同时, ErrorAll 信号还被送到动态电压调节模块, 产生用于指导电压生成模块工作的电压升高信号 VOL_INC 和电压降低信号 VOL_DEC。

原地监测单元监测到时序错误后, 需要一个额外的时钟周期进行错误纠正。因此在系统中采用了门控时钟的方法, 当 ErrorAll 为高电平时, 三级流水线乘法器主电路的时钟停一个周期用于纠正错误。保证乘法运算的输出结果 Mul_out[7:0]为正确的逻辑值。

该 AVS 系统在 65nm CMOS 工艺下进行了设计和后仿真验证, 标准工作电压是 1.2V。图 8-18 所示为 $T=25^{\circ}\text{C}$, FF 工艺角下, $f=300\text{MHz}$ 时系统动态电压调节效果图。系统工作之初时序比较宽松, 监测单元不会监测到时序错误, 16 个时钟周期后开始产生 VOL_DEC 信号以降低供电电压。电压的降低导致了时序的紧张, 此时, 监测单元监测到时序错误, 告警信号 ErrorAll 被拉高, 此信号在时钟上升沿被动态电压调节模块采样得到后产生电压升高信号 VOL_INC, 电压生成模块将电源电压升高 0.01V, 同时 AVS 系统利用门控时钟的方式将三级流水线主电路的时钟停一个周期, 对时序错误进行纠正。当电源电压为 0.87V 时, 最小系统时序正确。16 个周期内动态电压调节模块没有采样到时序错误, 电压降低 0.04V, 又产生了时序错误, 电压生成模块在动态电压调节模块产生的控制信号的指导下, 将电源电压以 0.01V 为步长逐步升高, 直到再次稳定在 0.87V。若 16 个周期内没有时序错误, 则再次将电源电压降低 0.04V。按照开环控制与闭环调节相结合的动态电压调节方法, 本系统在动态过程中充分释放了时序余量, 降低了动态功耗。

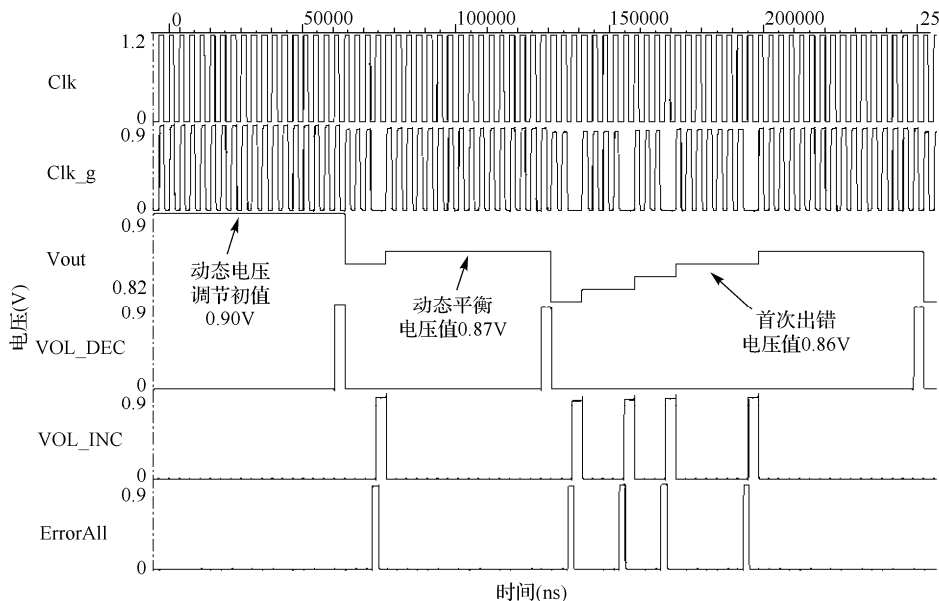


图 8-18 流水线乘法器 AVS 系统动态电压调节效果

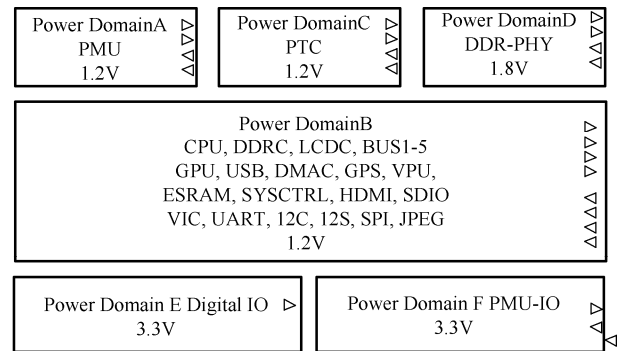
案例：SoC 芯片低功耗设计

本节以一款移动终端 SoC 芯片——SEP6200（关于 SEP6200 处理器的介绍，请参阅 2.8.5 节）为例，分析常用的芯片级低功耗技术的具体应用。SEP6200 是定位于手持视频播放设备、卫星导航产品的高性能低功耗处理器，在 TSMC 65nm 工艺下实现，包含 CPU 内核（Unicore II）、DDR、GPU、VPU 等模块，最高工作频率为 800MHz。

低功耗策略方面包括：采用多电压域设计，支持各电压域独立供电和掉电，时钟频率动态可调，支持 DVFS，支持低功耗模式，共有 3 种功耗模式可供切换：Normal 模式、Stop 模式、Sleep 模式。用户可根据具体工作场景对于性能的需求动态调节系统的工作频率、动态配置部分电压区域的工作电压或切断其供电电压，在满足性能需求的前提下，尽可能减小系统的功耗，从而延长电池供给设备的工作时间。

SEP6200 电压域的划分图如图 8-19 所示，共有 6 个电压域（Power Domain）：

- （1）Power Domain A：常开区，工作电压为 1.2V。电源管理单元（PMU，Power Manage Unit）放在此区，主要实现系统时钟控制和系统功耗控制的功能。
- （2）Power Domain B：数字核心区，工作电压为 0.9~1.2V，包含 CPU 内核、DDRC、LCDC、BUS1-5、GPU、USB、DMAC、GPS、VPU、ESRAM、SYSCTRL、HDMI、SDIO、VIC、UART、I2C、I2S、SPI、JPEG 等。该区域由外部 DC/DC 独立供电，支持 DVFS。
- （3）Power Domain C：备电区，RTC 的计时功能放在此区，当芯片进入 Sleep 模式时，此模块仍需供电。
- （4）Power Domain D：DDR-PHY 区，工作电压为 1.8V，DDR-PHY 模块是 DDR 控制器与 DDR 存储器的接口模块，时序要求较高，其工作电压符合 DFI（DDR PHY Interface）标准。
- （5）Power Domain E：数字 IO 区，工作电压 3.3V，数字核心模块与芯片管脚的接口模块。
- （6）Power Domain F：常开 IO 区，工作电压 3.3V，PMU 模块与芯片管脚的接口模块。



图中，◁和▷代表的是电平转换器和隔离单元

图 8-19 SEP6200 电压域的划分图

SEP6200 的各个电压域模块可以独立断电，模块之间插入了电平转换器和隔离单元。如图 8-20 所示，各个电源模块都由外部独立的电源供电，当需要关闭某模块时，配置 GPIO，通过不使能该模块的外部低压线性稳压源（LDO，Low Dropout Regulator）来使其掉电。当所有模块全部掉电（RTC 模块除外）后，通过长按键或者 RTC 唤醒信号使能外部 LDO 使其恢复供电。

除了提供用户自定义的时钟门控和电源门控，SEP6200 还提供几种低功耗模式，通过配置模式寄存器，可以使芯片实现模式自动切换功能。SEP6200 共有 4 种功耗模式：Normal 模式、Idle 模式、Stop 模式、Sleep 模式，RTC_ONLY 状态，这几种功耗模式的功耗依次降低。

Normal 模式: 所有的电压域全部开启。系统可以全速运行, CPU 内核可运行在 800MHz, DDR 可工作在 400MHz, 系统可通过 PLL 输出时钟 (普通状态), 也可动态配置 PLL 分频值和倍频值等参数以改变频率, 还可旁路 PLL, 直接由外部晶振输出时钟 (低速状态)。

Idle 模式: CPU 空闲模式。当 CPU 内核暂时没有任务需要处理的时候, 用户可通过软件执行指令使 CPU 内核进入 STANDBY 状态, 此时, CPU 内核进入自身的时钟门控模式, 关闭 CPU 内核中大部分的时钟, 其他模块的状态不变, 等待中断源触发退出此模式。

Stop 模式: 待机状态, 此模式下需要相对较长的唤醒时间。系统屏蔽所有模块时钟 (RTC 和 PMU 除外), PLL 进入 Power-Down 模式, 晶振选择性关闭 (可配置)。DDR 进入自刷新模式。其他模块的状态与进入 Stop 模式前的 Normal 状态下相同。

Sleep 模式: 手持设备的关机状态。在此模式下, DDR 进入自刷新模式。除去常开区之外的所有模块电源关断, PLL 进入 Power-Down 模式, 关闭晶振。

RTC_ONLY 状态: 芯片仅 RTC_Domain 工作, 其他模块掉电。

Normal 模式转换至 Stop 及 Sleep 模式的状态切换流程图如图 8-21 所示。

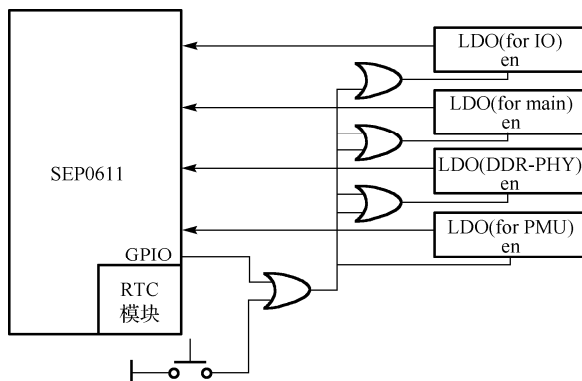


图 8-20 SEP6200 电源门控的实现

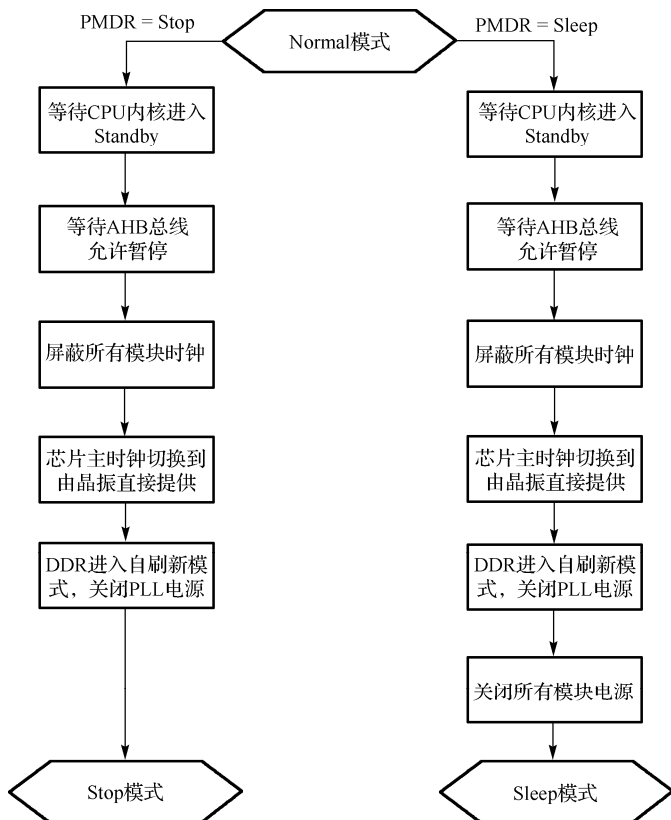


图 8-21 Normal 模式转换至 Stop 及 Sleep 模式的流程图

Normal→Stop: 当工作模式寄存器（PMDR）的值变为 Stop 时，判断 CPU 内核是否进入 Standby 状态，等待总线允许暂停，暂停总线，屏蔽各模块时钟，将时钟输入切换成晶振，PLL 进入 PowerDown 模式，选择性地关闭晶振。

Normal→Sleep: 当工作模式寄存器（PMDR）的值变为 Sleep 时，判断 CPU 内核是否进入 Standby 模式，等待总线允许暂停，暂停总线，屏蔽各模块时钟，将时钟输入切换成晶振，PLL 进入 PowerDown 模式，关闭非常开区电源，等待关电应答信号，选择性地关闭晶振。

Stop→Normal: 唤醒子模块收到唤醒信号，开晶振，开启锁相环，等待其稳定，打开所有被门控的时钟，等待 CPU 退出空闲状态，恢复总线，写工作模式寄存器的值为 Normal。

Sleep→Normal: 唤醒子模块收到唤醒信号，开晶振，开启锁相环，等待其稳定，控制外部电源电路给芯片内部上电（严格遵守模块上电顺序），等待上电完成的反馈信号，复位信号置位，打开所有被门控的时钟，恢复复位信号，恢复总线，打开隔离单元（Isolation），写工作模式寄存器的值为 Normal。

Stop 和 Sleep 模式转换至 Normal 模式的流程图如图 8-22 所示。

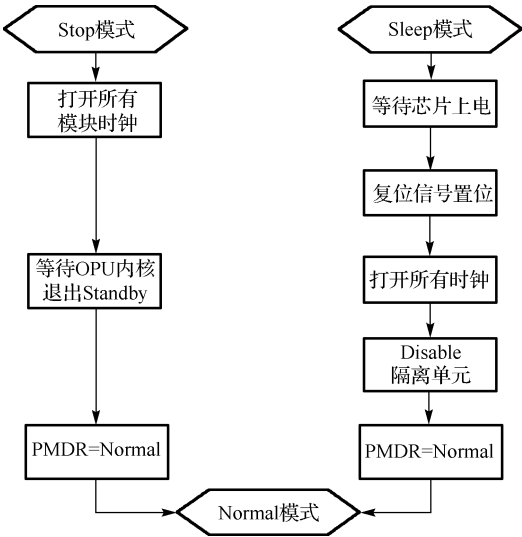


图 8-22 Stop 和 Sleep 模式转换至 Normal 模式的状态切换状态机

通过实验，我们对 SEP6200 微处理器各个电源域的功耗进行实际测量。功耗测量基于 SEP6200 的评估板 EVB6200。评估板板载两片 64M×16bit 的 DDR2 内存颗粒，4GB 容量的 NAND 存储器，并配有 7 英寸分辨率为 800×480 的 LCD 触摸屏，以及音频 Codec、以太网控制器、SD 卡接口、RS232 串口等常用外设。测试时，CPU 工作频率为 800MHz，内部 AHB 总线频率为 180MHz，DDR 频率为 400MHz，运行 Linux 操作系统，除 GPS、GPU 电源关闭外所有片上外设的电源与时钟都处于开启状态，所有板载外设都处于正常工作状态。表 8-2 列出了所测得的微处理器各个电源域功耗数据（不包含板载外设的功耗）。从功耗数据表中可以看出，功耗最高的 3 个电源域分别为 TOP 层（除 CPU 外的其他功能模块）、CPU 与 IO，分别占到了 SOC 总功耗的 40.75%、38.84%与 14.01%。但是在诸多电源域中，只有 A、C、F（参见图 8-19）需要始终供电，其他电源域可以在系统休眠时进行关电操作，也就是说系统休眠时 SEP6200 微处理器的功耗理论上可以降为工作时的 2%。

表 8-2 SEP6200 各电源域功耗数据

电源域	功能	功耗（mW）	百分比（%）
B	CPU	366	38.84

续表

电源域	功能	功耗 (mW)	百分比 (%)
A	PMU	18	1.91
C	RTC	—	—
B	TOP	384	40.75
D	DDRPHY	41.4	4.39
E	IO	132	14.01
F	PMU-IO	0.99	0.10
合计	—	942.39	—

8.3 嵌入式系统级低功耗设计方法

8.3.1 嵌入式系统级功耗优化技术介绍

总体上,嵌入式系统低功耗技术分为两大类:静态技术和动态技术。静态技术一般在嵌入式系统设计的初始阶段使用,假设系统的功能定义、工作模式以及运行流程等已知,而且在将来也不会改变。静态技术的实施效果在设计时即已确定,不会根据系统运行时的变化而有所调整,是一种根据大量设计时已知的系统信息进行功耗优化的技术。以下对常见的静态技术做一些简单的介绍。

(1) 通过软硬件划分实现的系统级功耗优化技术。软硬件划分技术从系统功能的高级抽象出发,试图寻求系统任务在软件与硬件之间的最优化划分与分配。在划分时以降低功耗为目标寻求最优解可达到构造低功耗系统的目的。原则上说,所有产生能耗的部件都可以在系统抽象级上进行描述,但功能的高度抽象会使对功耗的评估精度受到影响。

(2) 存储器静态能耗优化技术。①通过存储器层次的定制以及程序与数据在不同层次存储器中的合理布局实现降低存储器能耗的目标。②利用片上存储器(SPM, Scratch-Pad Memory)功耗相对片外存储器较低的特性进行能耗优化。详见第 5 章的 5.6.3 节。需要说明的是,早期的 SPM 功耗优化往往是将能耗较高的代码和数据(也就是访问频繁的代码和数据)在编译的时候就映射到 SPM 中,以降低系统总的能耗。但是最新的研究已经开始考虑在系统运行过程中动态地将不同代码和数据片段映射并搬运到 SPM,严格意义上来讲这已经不属于静态优化技术了。

与静态技术相比,动态技术则是系统在运行阶段充分利用工作负载的变化性来动态改变设备/芯片的工作模式、调整系统或策略的配置,从而降低系统功耗。因此相比静态技术的效果在设计时即已确定这点而言,动态技术在整个系统运行期间都能保持较好的功耗优化效果。动态技术所包含的内容也十分丰富,甚至在某些领域,静态技术也有向动态技术延伸发展的趋势。动态功耗管理(DPM, Dynamic Power Management)与动态电压调节(DVS, Dynamic Voltage Scaling)是系统级动态功耗优化技术中最具鲜明特点的两种主流技术,前者的研究对象是具有多种功耗模式的系统部件,后者的研究对象是支持运行时电压可调(同时也相应调整频率)的处理器。(读者需注意此处 DVS 与 8.2 节芯片级 DVS 的区别,芯片级 DVS 通常指基于硬件监测的电压调节,对用户是透明的;而此处嵌入式系统级 DVS 通常指基于软件调度的电压调节。)

由于研究对象的差异,DPM 与 DVS 具有明显的区别:DPM 一般针对系统部件进行研究,通过功耗模式的选择降低负载空闲时的功耗;DVS 一般针对处理器进行研究,通过工作电压的调节降低负载运行时的功耗。两者之间也存在着一致性:若将工作电压的连续或离散变化也看作广义的工作模式变化,则可将 DVS 纳入 DPM 的范畴之内。从该意义上出发,DVS 延伸了有效工作状态的定义,即包含

连续或者分散的电压值，使得在运行期间出现了若干能够在性能与功耗之间取得不同折中的工作状态。因此，DVS 与 DPM 彼此相互补充，相互配合，共同降低系统整体功耗。以下两小节分别具体介绍嵌入式系统的 DPM 和 DVS 低功耗技术。

8.3.2 动态电源管理 DPM

1. DPM 的基本原理与概念

动态功耗管理（DPM）是一种有效的系统级功耗优化技术。DPM 通过观察负载情况，在空闲时有选择性地系统或部件（Component）设置为低功耗模式以达到降低功耗的目的。DPM 得以发挥作用的两个基本前提是负载的非一致性以及可预测性。在嵌入式系统和大多数交互式系统中，负载的非一致性都非常普遍。由于应用与用户行为的相似性，负载的可预测性也可以在很大程度上得以满足。

在系统中，各部件的工作负载指来自系统的服务请求集合。当部件响应请求并提供服务时，称该部件为活动的（Active），否则称该部件为空闲的（Idle）。对部件的服务请求在时间轴上形成一个序列，如图 8-23 所示，每个箭头代表系统对某个部件的一次请求。空闲时间 T_{idle} 等于两次请求的间隔时间。以硬盘为例，当系统对硬盘产生读（写）请求时，硬盘接受请求并将数据从盘体送达系统（从系统写入盘体），在这一过程中硬盘处于 Active 状态；当硬盘完成读写操作后便处于 Idle 状态。

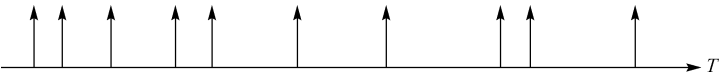


图 8-23 对部件的服务请求序列

系统部件的 Active 与 Idle 状态是从工作负载的角度进行划分的，而部件本身往往具有多个工作模式。一般来说，在 DPM 研究中，当部件不进行请求服务时，具有一个正常模式与若干个低功耗模式。当请求到达时，从某种工作模式到服务请求的延迟直接表征了该模式下的性能，延迟越小性能越好。正常模式的功耗最高，但在该模式下的部件性能也是最好的，一般可以认为没有延迟。低功耗模式的功耗比正常模式低，但性能也较差，并且模式的功耗越低其性能也越差。因此，部件的不同工作模式提供了性能与功耗的不同折中，使设计者可以根据系统的状况选择不同的工作模式以获得性能约束条件下的功耗最优。更为重要的是，部件在正常模式与低功耗模式间切换还会产生额外的能量损失，这就决定了当部件为 Idle 状态时，进入低功耗模式超过一定时间后所节约的能量才能大于模式转换时的能量损失，才能真正节约能耗。在 DPM 研究中，决定部件在 Idle 状态时是否值得进入低功耗模式以及何时进入的规则就称为策略（Policy）。从以上分析不难看出，DPM 研究一般只着眼于如何降低 Idle 状态时的功耗，而非 Active 状态时的功耗，这是 DPM 与 DVS 的一个显著区别。

部件在模式转换过程中所带来的能量损失决定了并不能在部件每次在 Idle 状态时都使其进入低功耗模式。损益平衡时间 $T_{break-even}$ （简称为 T_{BE} ）在 DPM 研究中用以表示能够获得能量节约的最小 Idle 时间，如图 8-24 所示，即当部件的 Idle 时间超过 T_{BE} 时，进入低功耗模式所节约的能量才能弥补转换带来的能量损失，才能真正节约能耗；而使部件在小于 T_{BE} 的 Idle 时间内进入低功耗模式反而会浪费能量。在正常模式

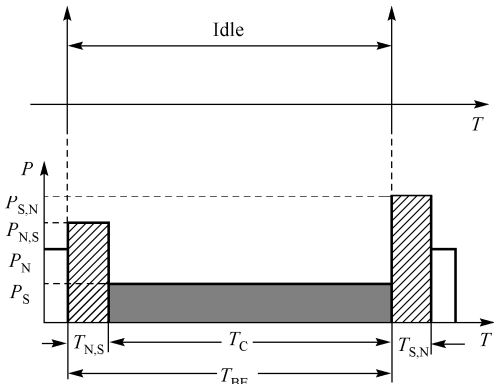


图 8-24 损益平衡时间的意义

下与不同低功耗模式间转换的能量损失并不相同,所以损益平衡时间是与某个具体的低功耗模式对应的。用 $T_{BE}(S)$ 来表示在低功耗模式 S 下的损益平衡时间。损益平衡时间是 DPM 中最重要的概念,直接反映了 DPM 的主要思想与原理,也是 DPM 策略做出决定以及对策略进行评估的重要依据。

在 DPM 策略范畴内,系统模型由一组相互作用的功耗可管理部件(PMC, Power Managable Component)和功耗管理器((PM, Power Management)两部分组成,其中 PMC 的工作模式由 PM 来控制。对 PMC 而言,其内部实现细节并不重要,而是将其看作黑盒子,专注于研究 PMC 和 PM 之间需要传递什么类型的信息及其信息量的大小。

2. 功耗可管理部件

PMC 被定义为完整系统内的一个原子模块。PMC 的概念具有一般性与抽象性,既可以小到芯片内部的一个模块,也可以大到一块电路板。在上面阐述 DPM 基本原理时所提及的系统部件,在 DPM 的研究范畴内即等同于 PMC。因此,损益平衡时间的概念也是针对 PMC 提出的,也只有 PMC 才具有损益平衡时间。

PMC 的基本特征是具有多个工作模式。这些工作模式都对应着不同的功耗与性能水平,其中性能水平一般通过从该工作模式到开始完成部件功能所经历的延时来标志,延时越短性能越高,而性能较高的模式一般功耗也较高。因此,PMC 的不同工作模式能够提供功耗与性能的不同折中。另外,PMC 也必须对外界提供模式选择的接口,以使 DPM 策略能够动态地选择 PMC 的模式,以获得性能约束条件下的功耗最优。PMC 的这一基本特征为 DPM 策略的实施提供了可能性。

PMC 的另一个重要特征是,由于工艺、结构及部件内部的异质构成等原因,PMC 在不同的工作模式间转换时会付出代价。在绝大多数情况下,代价意指能量损失及延迟。这里需要强调的是,在 DPM 研究中,该代价是不可忽略的。如果没有代价,系统可以使 PMC 在任何空闲的时候都进入低功耗模式以获得最大程度的能量节约,并且也不会影响性能,如此一来 DPM 就失去了存在的意义。因此,PMC 的这一重要特征体现了 DPM 研究的必要性。

3. 功耗管理器

功耗管理器是 DPM 策略的实施单元。PM 对 PMC、服务请求以及系统其他资源进行观察,将观察结果提供给 DPM 策略,并根据策略的决策对 PMC 进行控制,以降低 PMC 的功耗。

PM 实际上是一个控制器,既可以作为软件模块,也可以作为硬件模块,或者是软硬件的混合体存在于系统中。从实现层次上说,PM 的实现可以在电路层、驱动层、OS 层、应用层。从发展趋势看,在较高层次依赖软件实现的 PM 正成为主流。

图 8-25 说明了 DPM 研究中的 PM 结构。Observer 为观测模块,负责将 PMC 与系统的相关信息收集并提供给 Policy 模块。Policy 为策略模块,内部实现了 DPM 策略。Controller 为控制模块,根据 Policy 模块的决策向 PMC 发出模式控制的命令。

4*. DPM 策略

DPM 策略依赖于对 PMC 服务请求的观察与作用。策略与空闲时间序列的相互作用是对策略进行划分的最重要依据。在此依据下,将 DPM 策略划

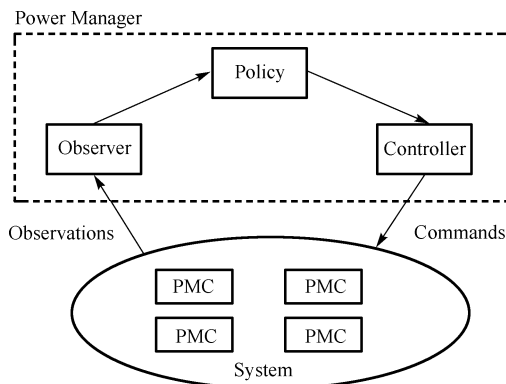


图 8-25 功耗管理器的结构

分为 4 类：超时策略，预测性策略，随机策略，改变服务请求的策略。前 3 种属于传统的 DPM 策略，第 4 种是相对较新的研究分支。

超时策略：超时策略（Timeout Policy）设置时间门限 T_{TO} ，一旦 PMC 持续空闲的时间超过 T_{TO} ，就使 PMC 进入低功耗模式。超时策略的前提假设为：如果 PMC 持续空闲的时间超过 T_{TO} ，则在剩下的空闲时间内使 PMC 进入低功耗模式所节约的能量能够抵销模式转换所带来的能量开销。早期所研究的是固定超时策略（Fixed Timeout）。顾名思义，固定超时策略设置固定的超时门限，在运行过程中不改变 T_{TO} 。固定超时策略应用广泛，开销极小，但超时门限固定这一先天特性使其很难在工作负载变化较大的环境中获得好的效果。针对固定超时策略的这一固有缺陷，产生了自适应超时策略（Adaptive Timeout）。自适应超时策略的主要思想是在运行时根据空闲时间以及其他系统因素的变化动态地调整超时门限，使策略更好地适应负载波动。超时策略的算法简单，易于通过超时门限的调节对策略效果进行控制，广泛应用于嵌入式系统中各类部件的功耗管理。但超时策略的弊端在于，无论空闲时间的长短如何，策略都要等待一段时间后才有可能进入低功耗模式，造成了等待时间的能量浪费。

预测性策略：DPM 预测性策略根据一定的规则预测部件下一次空闲时应该进入哪种工作模式才能最大程度地节约能量并在下次空闲时将部件切换到该模式。与超时策略相比，当预测性策略认为空闲时应该使部件进入低功耗模式时，不用等待超时门限结束，而是立刻进行切换，节约了超时等待所耗费的能量。预测性策略从本质上说是一种在线策略，因为策略在系统所有输入信息可用之前就必须依据历史信息做出判断。而工作负载在通常情况下是无法预知的，而且也是非稳态的，所以预测性策略能否自适应于负载的变化将变得十分关键。预测性策略都依赖于相邻空闲时间值之间存在的规律性，而在实际环境中，这种规律性未必能够体现，这也限制了现有预测性策略的效果。

随机策略：DPM 随机策略将部件请求以及服务过程建立为随机模型，较为精确地描述了系统的随机行为，选择进入可以带来功耗优化效果的部件模式，其本质是利用随机决策模型求解可以降低功耗的 PMC 模式控制算法。DPM 随机策略能获得功耗与性能的较好平衡，而且可以较为容易地调整平衡点，使策略具有极强的可控性。缺点是算法开销大，且假设空闲时间必须服从确定分布，使得随机性策略的可应用性也受到了较大的限制。

改变服务请求分布的策略：上述 3 种策略的实施效果很大程度上依赖于服务请求的分布。无论采用何种策略，都需要对请求分布进行分析，寻找其内在规律，从而离线或在线地选择更为合适的策略参数。然而在实际系统中：①请求分布往往并无规律可循或规律不断改变（如非稳态分布），因此较难准确分析；②部件服务请求的分布并不能很好满足 DPM 的要求，即相当一部分空闲时间小于损益平衡时间 T_{BE} ，则 PMC 在这些空闲时间内并不能进入低功耗模式。此时以上 3 种策略的实施效果会受到严重削弱。在满足性能要求的前提下改变部件服务请求的分布，从而使其规律容易捕捉且能较好地满足 DPM 要求，这样一类策略称为改变服务请求分布的 DPM 策略（DCP, Distribution-Change Policy）。

8.3.3 动态电压调节 DVS

1. DVS 基本原理

随着商用 CMOS 芯片电源供给技术的发展，在运行期间对处理器内核工作电压进行实时调节成为可能；而高效 DC-DC 电压转换器的出现也为处理器内核工作电压的动态调节提供了条件。伴随着处理器内核工作电压变化的是处理速度的变化。在嵌入式软实时系统中，任务的执行并无苛刻的时间要求，而只需要在应用规定的截止时间内执行完毕，截止时间决定了任务的紧迫程度。DVS 就是根据任务的紧迫程度对处理器内核电压进行动态调节，以达到任务响应时间和处理器低能耗之间的平衡的。

利用 DVS 技术降低处理器能耗基于这样一个事实：处理器的功耗与工作电压的平方成正比关系。

这与通过降低处理器频率来降低能耗具有本质区别，因为处理器功耗与频率成正比关系，任务运行时间与频率成反比关系，而任务运行所耗费的总能量为功耗与运行时间的乘积，因此降低处理器频率并不能减少运行任务所耗费的总能量（至少对于 CPU 总是处于 100% 利用率的情况下是如此）。

早期的 DVS 技术基于处理器的利用率进行设置，并没有考虑到运行任务的不同需求。后来针对实时系统提出了一些电压调节策略，举例如下。

2. DVS 策略模型

通过对一组任务的调度过程来阐述 DVS 策略模型。假设某个嵌入式处理器的工作电压能够在一定范围内连续调节，且内核程序需要处理 5 个相互独立的任务 T_a 、 T_b 、 T_c 、 T_d 、 T_e ，其中 T_a 、 T_b 是周期性的任务，另外 3 个任务则是间发性的，如表 8-3 所示。 T_a 、 T_b 的截止时间与它们的周期有关。每个任务在到达后可以立即被执行或者延迟执行，但是都必须在各自的截止时间到来之前执行完毕。

表 8-3 任务特征表

任务	到达时间	截止时间	周期	在 3.3V 下的执行时间
T_a	0	10	10	2
T_b	0	20	20	2
T_c	5	15	—	1
T_d	5	10	—	4
T_e	11	18	—	1

假设系统最大的可供电压为 3.3V，在该电压下的功耗被标准化为 1W。由 CMOS 器件特性可知，供给电压的降低将会导致电路延迟的增加；更精确一点，电路延迟的表达式为：

$$k*[V_{dd}/(V_{dd}-V_t)^2]$$

式中， k 为常数， V_{dd} 为工作电压（V）， V_t 为晶体管的阈值电压（V），此处不包含近阈值和亚阈值区域。

假设 V_t 的典型值为 0.8V。显然，当没有应用任何功耗降低技术时，系统的功耗为 1W。在对 DVS 调度技术进行说明的过程中，将其与 DPM 策略中的预测关闭技术进行了比较。当使用预测关闭技术时，假设系统完全预知工作负载的空闲时段，即处理器一旦进入空闲状态就立即将其关闭，从而使得该技术能够对系统功耗达到最大程度的优化。DVS 策略应用的最终目的在于满足各个任务截止时间的同时使得系统功耗最小化。任务调度过程采用了 EDF（Earliest-Deadline-First）调度机制。

图 8-26 显示了预测关闭技术和 DVS 方法的调度结果。

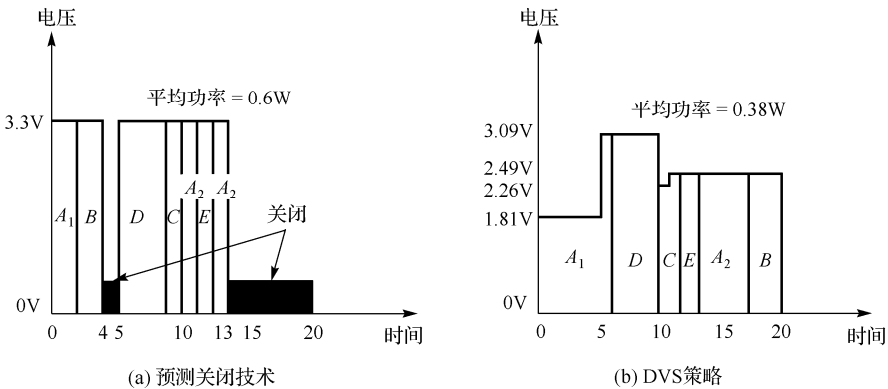


图 8-26 预测关闭技术和 DVS 策略的应用效果图

在系统预测关闭技术下，系统的工作电压一直为 3.3V。所有任务在[0,4]、[5,13]时间段内执行完毕，而处理器在 [4,5]、[13,20] 时间段内将被关闭，然后再为下一个周期性任务提供服务。处理器占空比是 60%，因此平均功耗为 0.6W。而在 DVS 应用过程中，系统的平均功耗为 0.38W，该值比预测关闭技术又降低了 $(0.6-0.38)/0.6=37\%$ 。

0.38W 只是在不知道间发性任务（即 T_c 、 T_d 、 T_e ）到达时间的情况下所能达到的最小功耗值。如果能够完全知道间发性任务的到达时间，则 DVS 最优策略就能够使得处理器在所有的时间内都维持在一个最低的电压水平，同时保证所有任务都满足截止时间的要求。在图 8-26 中，如果系统能够预知 T_c 、 T_d 、 T_e 的到达时间，则[0,20] 时间段内的最优电压为 2.48V，该电压值所对应的处理器速度为最大速度的 60%，即 $([3.3/(3.3-0.8)]^2 : [2.48/(2.48-0.8)]^2)$ ，该运行速度也导致系统的平均功耗降为 0.38W。显然，这个功耗平均值也对应着在不知道间发性任务到达时间的情况下系统功耗所能达到的最小边界值。

8.3.4 动态电压频率调节 DVFS

在许多处理器中，应用程序可能需要实时或者非实时地操作。一般来讲，在实时应用程序中，DVFS 技术能极有效地减少系统能量消耗且能满足系统的性能需求。

在 CMOS 电路中，单个任务的能量消耗跟供电电压和处理器频率的关系如下：

$$E = CV^2f_{clk}T$$

式中， V 是供电电压， C 是每个时钟周期的负载电容， f_{clk} 为时钟频率， T 为任务的总执行时间。因此，减少供电电压和时钟频率能极大地降低能耗。但是，减少供电电压和时钟频率却带来了相应的电路延迟。

DVFS 技术在能量消耗和执行时间上有一个权衡，所以 DVFS 需要选择合适的处理器电压和频率，在不影响处理器性能的前提下，更有效地减少它的能量消耗。

图 8-27 阐述了在实时应用中 DVFS 技术的基本原理：任务 W 的截止时间为 T_2 ， V_1 为 W 运行在高频时的电压，任务完成时间为 T_1 ， S_1 为 W 运行在高频时 CPU 的空闲时间，此时间段内 CPU 处于空闲状态且不消耗能量， V_2 为任务运行在低频时的电压，任务完成时间为 T_2 。假设 $V_2=V_1/2$ 、 $f_2=f_1/2$ 且 $T_2-T_1=T_1-T_0$ ，则根据公式 $E=CV^2f_{clk}T$ ，在降低供电电压的情况下能降低 75% 的能耗（这是理想情况，没有考虑关闭 CPU 的额外能量开销）。

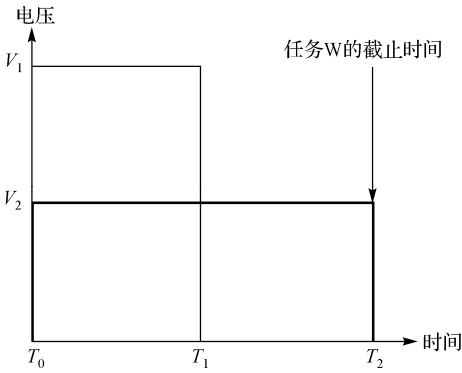


图 8-27 DVFS 技术的基本原理

在一个嵌入式系统中要实现 DVFS 功能，除了需要处理器具有频率或电压调节的功能外，还需要精确的负载预测，这样才能在不影响系统性能的前提下最大化地降低系统功耗。处理器在 DVFS 算法下的整体框架如图 8-28 所示。通过 CPU 负载监测模块记录 CPU 的使用情况，计算出 CPU 的利用率，然后根据预设的策略，决定是否调频。如果需要调频，则硬件 DVFS 控制模块通过时钟和电源接口来分别调节处理器的时钟频率和供电电压。

根据约束类型、调节粒度和策略制定方式，DVFS 可分为如下 3 类。

(1) **实时 DVFS 和非实时 DVFS**：根据约束类型，DVFS 技术可以分为实时和非实时。实时系统的任务都具有一个截止时间，根据截止时间的严格性，实时又可以进一步分为硬实时和软实时。硬实时是任务的截止时间必须能得到完全满足，否则可能造成不可预料的后果。在硬实时环境中应用 DVFS 算法已经是一种成熟的技术。例如，对于操作系统调度器，每个任务的开始时间、到达时间和负载都

是固定的, 因此需要硬实时 DVFS 算法。软实时任务虽然需要一个截止时间, 但实时任务偶尔违反折中需求对系统运行不会造成严重影响, 如监测系统和信息采集系统等。软实时和非实时没有明确的截止时间, 负载大小是在程序运行时预测的, 因此在满足性能和服务质量的前提下需选择最小的电压和频率。

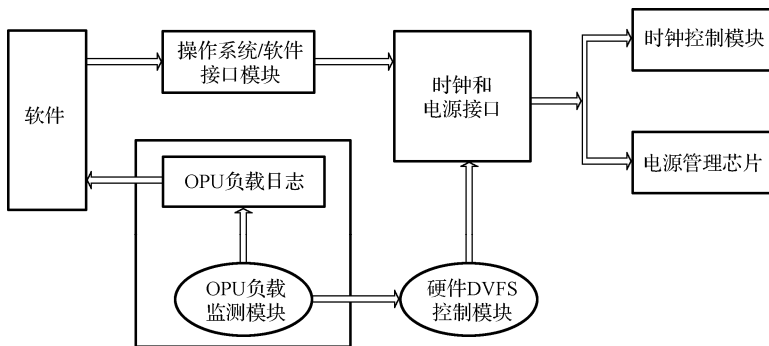


图 8-28 CPU 在 DVFS 算法下的整体框架图

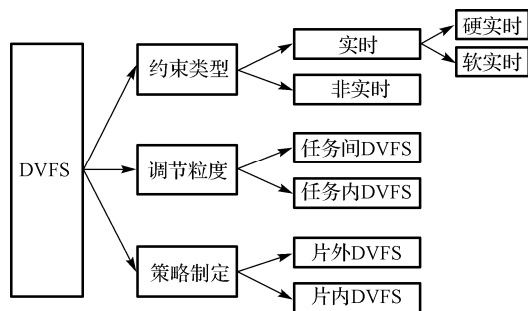


图 8-29 DVFS 的分类

(2) **任务间 DVFS 和任务内 DVFS:** DVFS 根据调解粒度可以分为两类: 任务间和任务内。任务间的调节是粗粒度的, 工作在操作系统级; 而任务内的调节是细粒度的, 工作在单任务程序中。多任务硬实时应用适用于任务间的调节粒度, 操作系统给每个任务布置合适的时间且保证所有的任务在截止时间之前完成。而软实时适用于任务内的调节粒度, 在软实时系统中不存在空闲时间的情况, 每个时间片都能得到充分的利用, 因此极大地提高了能量利用率。

(3) **片外 DVFS 和片上 DVFS:** 片外 DVFS 算法在任务执行之前就已确定了 CPU 的目标频率。片外 DVFS 算法的一个典型例子是硬实时任务间的电压调节, 此时任务的负载通常通过仿真而得到。通常做法是: 给定任务的截止时间和负载信息, 用公式描述一个整数线性或非线性程序问题, 对每个任务计算一个合适的 CPU 频率。片上 DVFS 算法是在任务执行期间基于历史信息决定 CPU 的频率, 事先并不知道任务将如何运行。因为片上 DVFS 算法是基于历史信息的, 所以负载预测的精确度极大程度地决定了 DVFS 算法的有效性。例如, 任务内的 DVFS 算法就是一种片上 DVFS 算法。

DVFS 技术的一个关键问题是如何准确有效地预测 CPU 的负载。在预测 CPU 负载时, 必须考虑两个问题: 算法复杂度和预测的精确度。如果算法过于复杂, 额外的算法开销可能会大于 DVFS 技术带来的节能, 得不偿失; 如果负载预测偏差较大, 那么 DVFS 技术也就毫无意义。因此, 在预测 CPU 负载时, 应该权衡算法复杂度和预测的精确度, 提出一种合理的负载预测机制。

案例：整机系统级低功耗设计

本节将介绍基于 SEP6200 嵌入式微处理器的 3G 电子血压计的整机系统级低功耗设计方案（关于 SEP6200 处理器，请参阅本书的 2.8.5 节。关于 3G 电子血压计的功能请参阅本书第 6 章的案例，虽然该案例是基于 SEP4020 处理器的，但是 SEP6200 版本的整机功能与之相类似），包括硬件电路设计、底层驱动、操作系统低功耗适配与针对具体应用的上层低功耗策略等。

本案例中的 3G 智能电子血压计以 SEP6200 微处理器为主控芯片，搭载 128MB 的 DDR2 内存、256MB 的 NANDFLASH、3G 通信模块、血压测量模组、4.3 寸 LCD 触摸屏、键盘、SD 卡接口等。电源采用市电与电池两种供电方式，增加了系统的便携性。系统平台架构如图 8-30 所示。

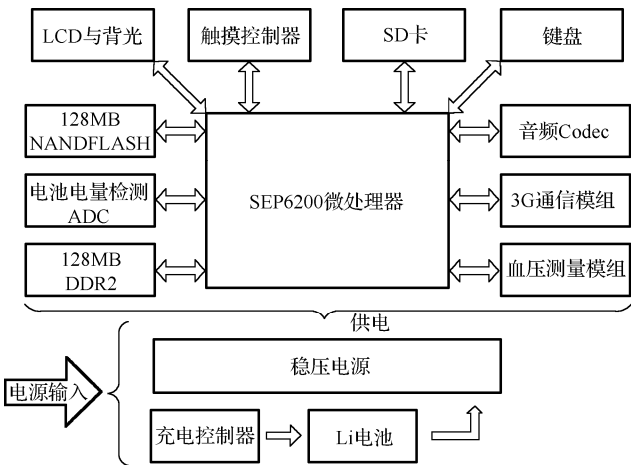


图 8-30 系统平台架构图

1. 电源系统设计

电源是嵌入式系统中不可缺少的重要组成部分，电源设计的好坏直接决定了系统设计的成败。当前的嵌入式系统往往由多个模块组成，而不同的模块对电源的要求也各不相同，如电压、电流、纹波、效率、体积等。因此需要针对系统中的模块，合理地设计电源，使系统的电源在一定成本下达到最高效率，有效降低系统的功耗。

电源设计的低功耗主要从两方面考虑：①芯片选型；②电源结构。

按照电源调整管的工作状态来分，直流稳压电源可以分为两大类——线性电源和开关电源。调整管工作在线性状态的称为线性电源，工作在开关状态的称为开关电源。线性电源可以细分为两种，一种是普通线性稳压器，另一种是低压差线性稳压器（LDO，Low Dropout Regulator）。开关电源也可以细分为两种，一种是电容式 DC-DC 转换器，即常说的电荷泵；另一种是电感式 DC-DC 转换器，即通常所说的 DC-DC 转换器。表 8-4 列出了上述 4 种电源芯片的比较。

表 8-4 4 种电源芯片的比较

电源类型	普通线性稳压器	LDO	电荷泵	DC-DC		
				Buck	Boost	Buck-Boost
效率	低	中	中到高	高	高	高
成本	低	低	中	高	高	高
尺寸	小	最小	中	大	大	大

续表

电源类型	普通线性稳压器	LDO	电荷泵	DC-DC		
				Buck	Boost	Buck-Boost
纹波噪声	小	最小	小	大	大	大
设计难度	低	低	中	高	高	高
输出电流	中	小	低	大	大	大
热量管理	差	中	好	最好	最好	最好
局限性	不能升压	不能升压	无	不能降压	不能升压	无

从表中可以看出，线性电源具有简单、尺寸小、噪声低、廉价等诸多优点，但是效率不高。开关电源与之相反，具有高效率的特点，但在其他方面较线性电源没有优势。

在嵌入式移动平台中大多是电池供电，优先考虑转换效率可以降低电源损耗，使电池的使用时间更长，因此无论升压、降压都可以选用 DC-DC 稳压器。特别是在电池供电系统电源负荷变化较大的应用中，可以选择 PFM/PWM 自动切换控制式的 DC-DC 稳压器。PWM 的特点是噪声低，满负载时效率高。PFM 静态功耗小，在低负荷时可改进转换器的效率。系统在重负荷时由 PWM 控制，在低负荷时自动切换到 PFM 控制，这样就能够兼顾轻重负载的效率。

但是由于 DC-DC 稳压器高噪声的缺点，不能作为对噪声有严格要求的设备的电源，在嵌入式系统中这些设备常见的有 PLL、RTC、音频 Codec、ADC/DAC 与基准源等，只能选用线性电源。此外在成本控制严格的应用中，选用线性电源，特别是选用 LDO 等效率相对较高的电源芯片也是很普遍的。也有很多应用做出折中选择，采用 DC-DC 与线性电源混合的方式，对于电流需求较高的外设采用 DC-DC 稳压器供电，线性电源仅对低功率设备供电，由于线性电源输出电流低，损耗也就低，这样在节约成本的同时也可以达到不错的转换效率。

常用的电源结构有 3 种：①级联降压；②独立降压；③混合降压。

级联降压指的是先把较高电压通过第一级稳压器降至一个较低的中间电压，再通过二级稳压器转换出系统其他模块所需的更低的电压，然后逐次降低到所需的最低电压。在独立降压结构中系统中只有一级稳压器，利用多个稳压器把输入电压直接转换成系统所需的各种电压。混合降压就是把前两种拓扑结构结合起来。

SEP6200 微处理器采用了多源多压技术。这种技术可以有效降低芯片的功耗，但同时也增加了系统电源设计的复杂度。表 8-5 列出了 SEP6200 微处理器芯片的电源引脚。从表中可以看出仅微处理器的电源引脚多达 11 种。此外还有诸多的外部模块需要供电，表 8-6 列出了外部模块的电源。

表 8-5 SEP6200 微处理器芯片电源引脚列表

名称	说明	供电电压	是否常开
VDD_ARM	CPU 电源引脚	1.2V	否
VDD	Top 层电源引脚	1.2V	否
VDDQ	DDR-PHY 电源引脚	1.8V	否
VDDPST	IO 电源引脚	3.3V	否
PLL_VDD	主晶振数字电源引脚	1.2V	是
PLL_AVDD	主晶振模拟电源引脚	1.2V	是
VDD_PMU	PMU 电源引脚	1.2V	是
VDDPST_PMU	PMU-IO 电源引脚	3.3V	是
VDD_RTC	RTC 电源引脚	1.2V	是
CLK32K_VDD	实时晶振数字电源引脚	1.2V	是
CLK32K_AVDD	实时晶振模拟电源引脚	1.2V	是

表 8-6 外部设备电源列表

名称	说明	供电电压	是否常开
VDD_NAND	NAND 芯片电源	3.3V	是
VDD_DDR	内存芯片电源	1.8V	是
VDD_LCD	液晶屏电源	3.3V	是
VDD_LED	液晶背光源	12V	否
VDD_TS	触摸屏芯片电源	3.3V	是
VDD_SD	SD 卡接口电源	3.3V	是
VDD_CODEX	音频 Codec 电源	3.3V	否
VDD_ADC	电池容量 ADC 电源	3.3V	否
VDD_CDMA	3G 通信模组	3.3~4.2V	是（有卡）/否（无卡）
VDD_XYJ	血压测量模组	5V	否

为每一个电源引脚或者设备设计一路电源显然是不合理的，过多的电源芯片不仅大幅增加了产品的尺寸与成本，也带来了更多的电源转换损耗与静态功耗。良好的电源设计不仅需要采用合理的结构与优秀的芯片，也要对系统进行良好的电源分区，合理分配多个模块采用一路电源供电，以最少的电源数量达到最好的设计要求。通过分析可以发现，在系统运行中与休眠后，需要将不需要工作的模块电源关闭。因此在电源分区时，可以按照常开与否将电源分为两类，每一类内的电源可以按照电压大小进行合并，相同电压由一路电源进行供电，从而达到精简的目的了。图 8-31 给出了整机设计的电源结构框图。

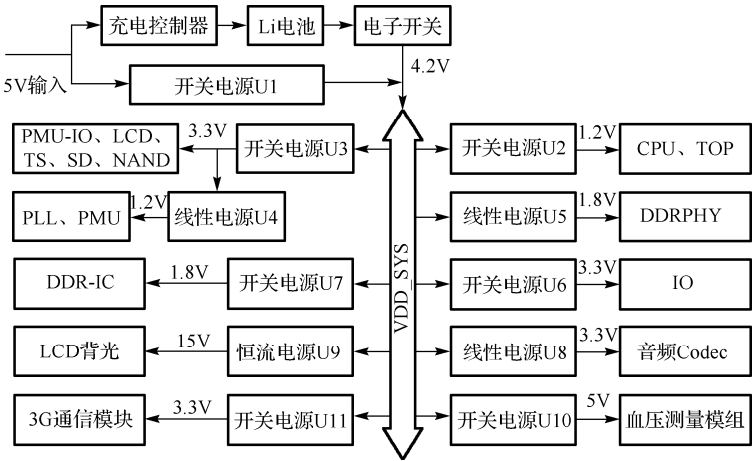


图 8-31 电源结构框图

2. 整机低功耗模式设计

SEP6200 处理器提供了 4 种工作模式：Normal、Idle、Stop 和 Sleep。由于 Stop 模式下并不进行掉电操作，仅仅将时钟关闭，在 65nm 工艺时，芯片的静态功耗已经是不可忽视的一部分，因此仅仅关闭时钟并不能满足系统对于低功耗的需求，所以在本方案的设计中不准备采用 Stop 模式。系统的工作模式及其转换关系如图 8-32 所示。

Idle 模式是 SEP6200 微处理器的一个轻量级低功耗工作模式，旨在降低 CPU 的功耗。在该模式下，CPU 处于 Bypass 状态，大部分时钟停止工作，但监控中断请求的部件始终处于活动状态，监控返回 Normal 的事件发生。由于 CPU 只是时钟停止工作，但不掉电，CPU 的寄存器的数据并不丢失，其他片上或者片外设备的工作状态在 Normal 和 Idle 的模式切换中并不发生改变，如 Timer、中断控制

器、DDRC、DMAC 和 LCDC 等在 Idle 模式中都处于正常的工作状态, 因此 Normal 和 Idle 之间因为状态切换而付出的代价很低。此外, 这两个模式的切换过程也比较简单, 不需要对系统软硬件工作状态进行保存和恢复, 只需要在系统没有任务运行时, 设置寄存器 PMDR 的最高位为 1 即可进入 Idle 模式。CPU 在接收到 Idle 切换指令后, 就会在运行完流水线中的指令以后立即进入 Idle 模式。Idle 模式的退出则依赖于有效的中断事件, 一旦有片内或者片外的中断请求发生, 且相应的中断信号并未屏蔽, CPU 就会立刻退出 Idle 状态进入 Normal, 从上次中断的地方继续执行。

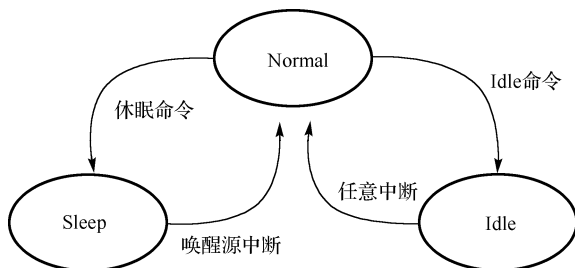


图 8-32 系统的工作模式转换关系

在 Linux 系统中, 存在一个进程号 (PID, Process Identifier) 为 0 的特殊进程, 被称为 Idle 进程, 或者 Swapper 进程。当 Linux 系统的进程就绪 (TASK_RUNNING) 队列为空, 没有其他进程要运行时, 内核将调度 Idle 进程运行。因此, 只要把 SEP6200 处理器 Idle 模式的切换逻辑放在 Idle 进程的关键函数 `cpu_idle` 中, 即可让 CPU 在空闲时安全而有效地进入 Idle 模式。当不存在大量计算任务和高负荷服务进程时, 系统处于 Idle 状态的时间会非常多。根据官方给出的 Linux 内核统计数据, 系统平均处于 Idle 状态的时间占到了总时间的 60%~80%。因而系统 Idle 状态的实现对于降低系统功耗, 尤其是 CPU 的功耗, 有着非常积极的作用, 是整个系统级低功耗技术的重要组成部分。

在一定时间内如果没有用户使用请求, 则可将系统切换到 Sleep 状态, 关闭大部分电源, 可以最大限度地减少系统不必要的功耗。休眠是系统除关机 (也就是整机掉电) 外最为省电的一种状态。SEP6200 从 Normal 进入 Sleep 状态, 会首先关闭 CPU 及所有模块的时钟, 包括 Timer、中断控制器、DDRC、DMAC 和 LCDC 等系统运行必需的模块, 然后关闭 PLL, 接下来 PWRON_MAIN、PWRON_ARM 和 PWRON_IO 三个引脚输出变低使外部电源关闭, SoC 系统除常开区外的所有模块都掉电。此时仅仅保留部分重要电路工作, 比如 RTC、PMU, 这些部件主要负责监视预先设置好的唤醒事件。一旦发生该事件, 则重新开启 PLL 和外部电源, 并发出系统复位信号, 逐步进行唤醒操作。

Sleep 模式的进入同 Idle 类似, 通过设置寄存器 PMDR 的低两位为 0x01 来完成。为了系统在唤醒时能够顺利恢复到正常工作, 需要软件在系统进入 Sleep 前做两件事情: 首先, 要设置好系统的唤醒源, SEP6200 唤醒源包括 Wakeup 按键、RTC 中断和外部中断; 其次, 由于在 Sleep 模式下大部分模块处于掉电状态, 这些模块的寄存器会在进入休眠后丢失, 所以需要将这数据保存到 DDR 内存中, 并设置为自刷新模式, 以免内部的数据丢失, 从而在唤醒时能够顺利从内存中恢复这些数据。

唤醒过程实际是系统复位过程, 系统的引导装载程序 (BootLodaer) 有一个很重要的职责——判断系统是从睡眠状态恢复还是执行复位操作。若是后者, 则进行正常的重启操作序列; 若是唤醒, 则直接跳转到内核唤醒程序入口, 从 DDR 内存中恢复系统工作现场, 回到系统睡眠前的运行状态。可见系统休眠唤醒要比 Idle 状态的切换复杂得多, 系统要完成大量数据、设备的保存与恢复操作, 同时带来的状态切换的代价也更高, 频繁的系统休眠和唤醒操作不仅不会降低功耗, 反而会增加大量系统时间和电能的消耗, 因此何时进入休眠状态也是系统休眠的一个重要问题, 需要通过一个好的休眠唤醒策略来调节。

3. 整机低功耗模式的测试结果

利用 Linux 内核的进程调度机制与 Idle 进程，当系统中没有其他进程要运行时，Idle 进程被运行。在 Idle 进程中，通过将 CPU 配置为 Idle 模式，关闭 CPU 的大部分时钟，达到节约功耗的目的。CPU 功耗测试结果（注意，仅仅是 CPU 的功耗测试，不包括其他模块的功耗）如图 8-33 所示，其中 NOP 为优化前无任何操作直接返回的功耗，LOOP 则仅进行 50 个 for 循环但不配置 CPU 进入 Idle，IDLE 则为配置 CPU 进入 Idle 模式的功耗。此时系统处于开机状态，上层应用程序已经加载完毕，但没有用户操作。通过 Top 命令查看，Idle 进程的占用率达到了 98%~99%。可见对于不存在大量计算任务和高负荷服务进程的应用，将 CPU 配置进入 Idle 模式可以大量节省功耗。

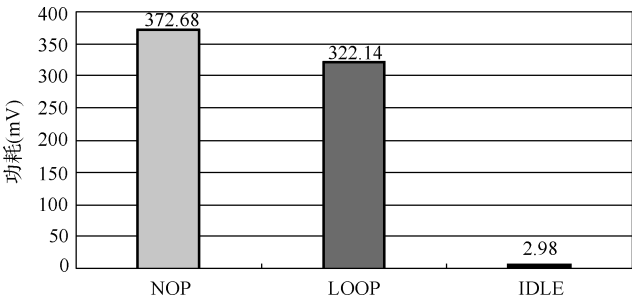


图 8-33 CPU Idle 模式功耗

在对 CPU 的功耗进行测量之外，还对系统其他电源域，尤其是与 CPU 紧密相关的 TOP 层与 DDR 内存的功耗进行了测量。经过测量发现，在仪器的可测量精度内其功耗保持不变。这是由于 CPU 中 Cache 的存在，即使不使 CPU 进入 Idle 状态，Cache 的高命中率中也可以降低 CPU 对总线与内存发起的访问次数。

系统休眠是除系统关机外最为省电的一种状态。进入休眠后，微处理器内部模块时钟被关闭，LCD 背光处于关断状态，DDR 内存进入自刷新模式，并且可以选择性地对系统进行关电操作，达到更低的休眠功耗。图 8-34 中列出了系统全速运行时、休眠但不进行关电操作与休眠并且进行关电操作的功耗。可以看出，系统休眠时进行不关电与关电操作所达到的休眠功耗分别为系统全速运行时的 19.2%和 3.1%。进行关电操作可以得到更低的休眠功耗。

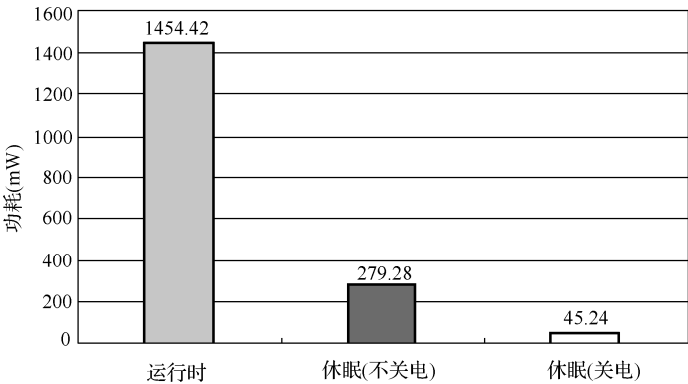


图 8-34 系统运行与休眠时功耗

图 8-35 列出了两种休眠情况下的系统功耗分布。进入休眠后不关电时，SEP6200 微处理器的 TOP 层和电源损耗占据了主要的部分。可以看出，此时时钟源关闭，TOP 虽然不会产生动态功耗，但其静

态功耗依然很大。电源损耗与运行时相比所占的比重更大,这是由于所有电源芯片依然处于工作状态,当电源芯片的输出电流降低时,其转换效率也会随之降低。此处 CPU 并没有产生功耗,是由于在休眠进入时将 CPU 配置为 Idle 模式。进行关电操作时,整个系统的功耗由外设、DDR 内存与电源损耗 3 部分组成。DDR 内存处于自刷新模式,而外设部分都是在整个系统运行过程中不可掉电的部分,因此若要进一步降低休眠功耗只能从电源入手,选用在低输出电流时依然具有较高转换效率的电源转换芯片。

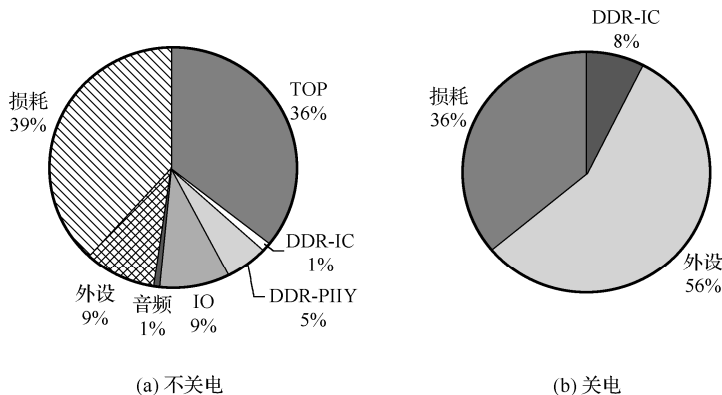


图 8-35 系统休眠时的功耗分布

通过数据分析可见, 45mW 的休眠功耗基本达到了系统休眠功耗的极限值。按照 3600mAh 容量的 Li 电池计算, 该应用平台可以实现 300 多小时的待机时长, 满足系统的设计需求。

思考题

1. 芯片的静态功耗和动态功耗二者哪个是低功耗优化的重点? 常见的几种低电压方法分别降低的是哪种功耗?
2. 将以下几种常见的低功耗策略的功耗收益按大小进行排序: 时钟门控、电源门控、动态电压调节。

扩展阅读

- [1] Weiwei Shan and Zhipeng Xu, "Timing Error Prediction based Adaptive Voltage Scaling for Dynamic Variation Tolerance" [C], 2014 IEEE Asia Pacific Conference on Circuit and Systems, 2014, pp. 739-742.
- [2] Kathryn Wilcox and et. al, "A 28nm x86 APU Optimized for Power and Area Efficiency"[C], ISSCC Dig. Tech. papers, 2015, pp. 84-86.
- [3] Shidhartha Das, David Roberts, Seokwoo Lee, et al, "A self-tuning DVS processor using delay-error detection and correction" [J], IEEE Journal of Solid-State Circuits,, 2006, Vol. 41, NO.4, pp. 792-80.
- [4] S. Das, et al, "Razor II: In situ error detection and correction for PVT and SER tolerance" [J], IEEE Journal of Solid-State Circuits,, 2009, Vol. 44, NO.1, pp. 32-48.
- [5] Weiwei Shan, Haolin Gu, and et.al, "An improved timing monitor for deep dynamic voltage scaling system" [J], IEICE Electronics Express, 2013, Vol. 10. NO. 6, pp. 1-7.
- [6] Elgebaly M, Sachdev M. "Variation-aware adaptive voltage scaling system"[J]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems , 2007, 15(5): 560-571.
- [7] Gammie G, Wang A, Mair H, et al. "SmartReflex power and performance management technologies for 90 nm, 65 nm, and 45 nm mobile application processors"[J]. Proceedings of the IEEE, 2010, 98 (2) : 144-159.

- [8] Drake A, Senger R, Deogun H, et al. A distributed critical-path timing monitor for a 65nm high-performance microprocessor[C]. ISSCC Dig. Tech. papers, 2007: 398-399.
- [9] Nakai M, Akui S, Seno K, et al. "Dynamic voltage and frequency management for a low-power embedded microprocessor"[J]. IEEE Journal of Solid-State Circuits, 2005, 40(1): 28-35.
- [10] Ikenaga Y, Nomura M, Suenaga S, et al. "A 27% active-power-reduced 40-nm CMOS multimedia SoC with adaptive voltage scaling using distributed universal delay lines"[J]. IEEE Journal of ISSCC Dig. Tech. papers, 2012, 47 (4) : 832-840.
- [11] Kim J, Horowitz M A. "An efficient digital sliding controller for adaptive power-supply regulation"[J]. IEEE Journal of Solid-State Circuits, 2002, 37 (5) : 639-647.
- [12] 陈春章, 艾霞, 王国雄. 数字集成电路物理设计[M]. 科学出版社, 2008.
- [13] 殷宏. 基于 Garfield 的 SoC 功耗估计与分析[D]. 硕士学位论文. 南京: 东南大学, 2004.
- [14] 赵梦南. 针对 SEP0611 DDR 控制器的功耗优化[D]. 硕士学位论文. 东南大学, 2012.
- [15] Keating M. Low Power Methodology Manual: For System on Chip Design[M]. Springer, 2007.
- [16] 黄少珉. 嵌入式存储系统 DPM 策略研究[D]. 博士学位论文. 东南大学, 2007.
- [17] 卜爱国. 嵌入式系统动态低功耗设计策略的研究[D]. 博士学位论文. 东南大学, 2006.
- [18] 戚隆宁. 基于负载分析的嵌入式系统动态功耗管理策略的研究[D]. 博士学位论文. 东南大学, 2007.
- [19] Gethin Norman, David Parker, Marta Kwiatkowska, et al. "Using Probabilistic Model Checking for Dynamic Power Management"[J]. Formal Aspects of Computing. 2005, 17(2):160-176.
- [20] Nathaniel Pettis, Le Cai, Yung-Hsiang Lu. "Dynamic Power Management for Streaming Data"[C]. In: Proceedings of the 2004 International Symposium on Low Power Electronics and Design. New York, NY, USA:ACM Press, 2004:62-65.
- [21] Matthias Eireiner, Stephan Henzler and et.al, "In-situ delay characterization and local supply voltage adjustment for compensation of local parametric variations" [J], IEEE Journal of Solid-State Circuits, 2007, Vol.42, NO. 7, pp.1583-1592.
- [22] K.A.Bowman, J.W.Tschanz, S.L.Lu, et al., "A 45 nm resilient microprocessor core for dynamic variation tolerance" [J], IEEE Journal of Solid-State Circuits, 2011, Vol. 46, NO.1, pp.194-208.
- [23] BullD., Das S, Shivashankar K.,Dasika G.S. et al., "A power-efficient 32 bit ARM processor using timing-error detection and correction for transient-error tolerance and adaptation to PVT variation" [J], IEEE Journal of Solid-State Circuits,, 2011, Vol. 46, NO.1, pp. 18-31.
- [24] Inyong Kwon, Seongjong Kim, et al., "Razor-Lite: a light-weight register for error detection by observing virtual supply rails" [J], IEEE Journal of Solid-State Circuits,, 2014, Vol. 49, NO.9, pp. 2054-2066.